
Design Patterns (II)

DA243, Object Oriented Animals

Design Patterns

- Design Patterns
 - Concepts which we can apply over and over
 - Last time
 - Iterator
 - Composite
 - Factory
 - Now more challenging patterns
 - Master these, and nothing will intimidate you!
-

Design Patterns

■ This time

- Decorator
- Visitor
- (and in passing we'll mention Strategy Pattern)

■ References

- “Design Patterns: Elements of Reusable Object-Oriented Software”

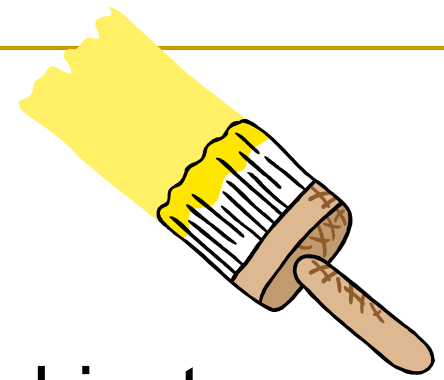
—E. Gamma, R. Helm, R. Johnson and J. Vlissides

- And also, “The Design Patterns Java Companion”

<http://www.patterndepot.com/put/8/JavaPatterns.htm>

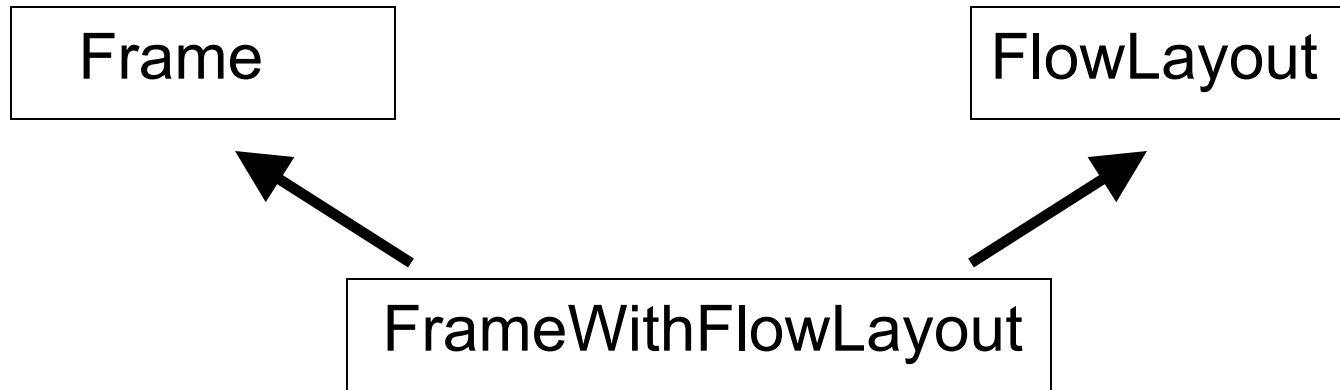
James W. Cooper

Decorator: Problem



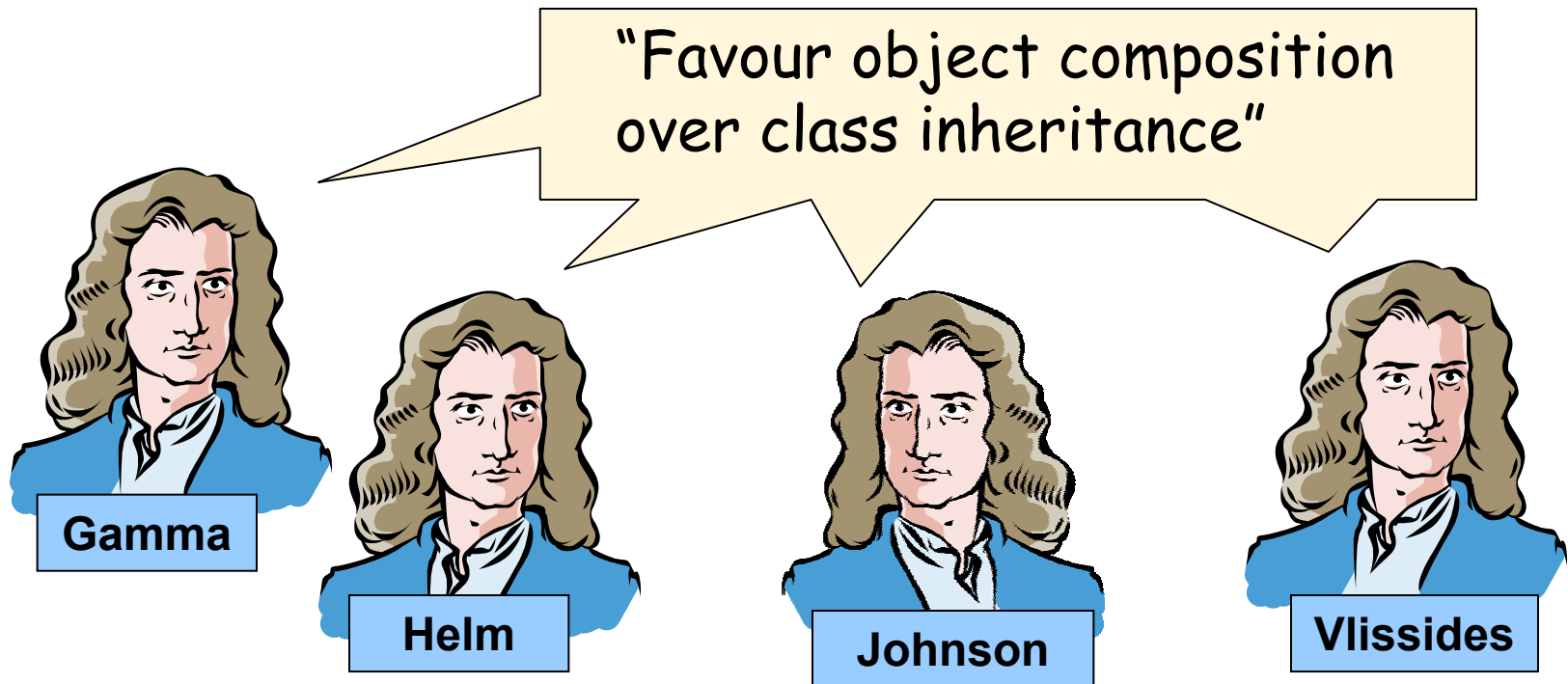
- Want to add functionality to one object
 - Could use inheritance
 - E.g., Suppose want to add layout to Containers
 - Then need `FrameWithBorderLayout`, `FrameWithGridBagLayout`, `FrameWithFlowLayout`, etc.
 - Also need `PanelWithBorderLayout`, `PanelWithGridBagLayout`, `PanelWithFlowLayout`, etc.
 - Etc.

Decorator: Problem



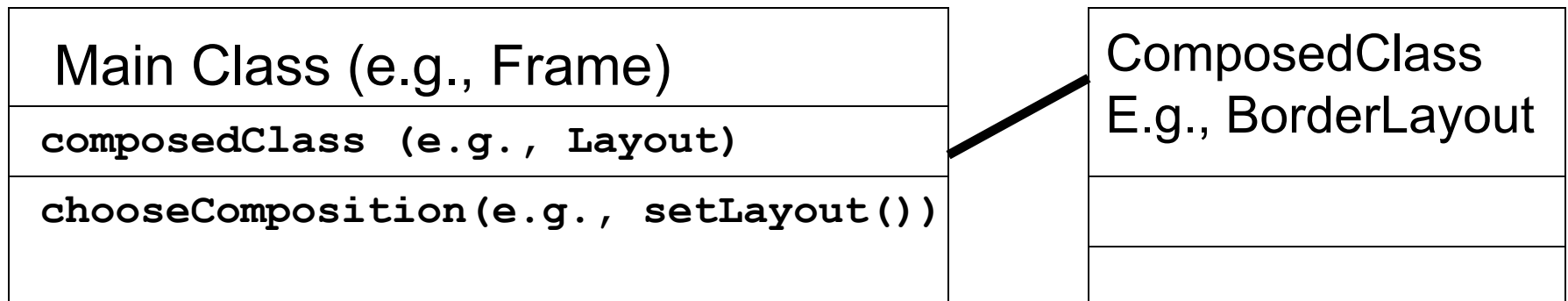
- Multiple inheritance helps reuse code, but
 - ❑ Still need many, many subclasses
 - ❑ Multiple inheritance is messy
 - ❑ Java doesn't support it anyway!
- Inheritance adds functionality to *all* objects, not just one

Solution: Composition



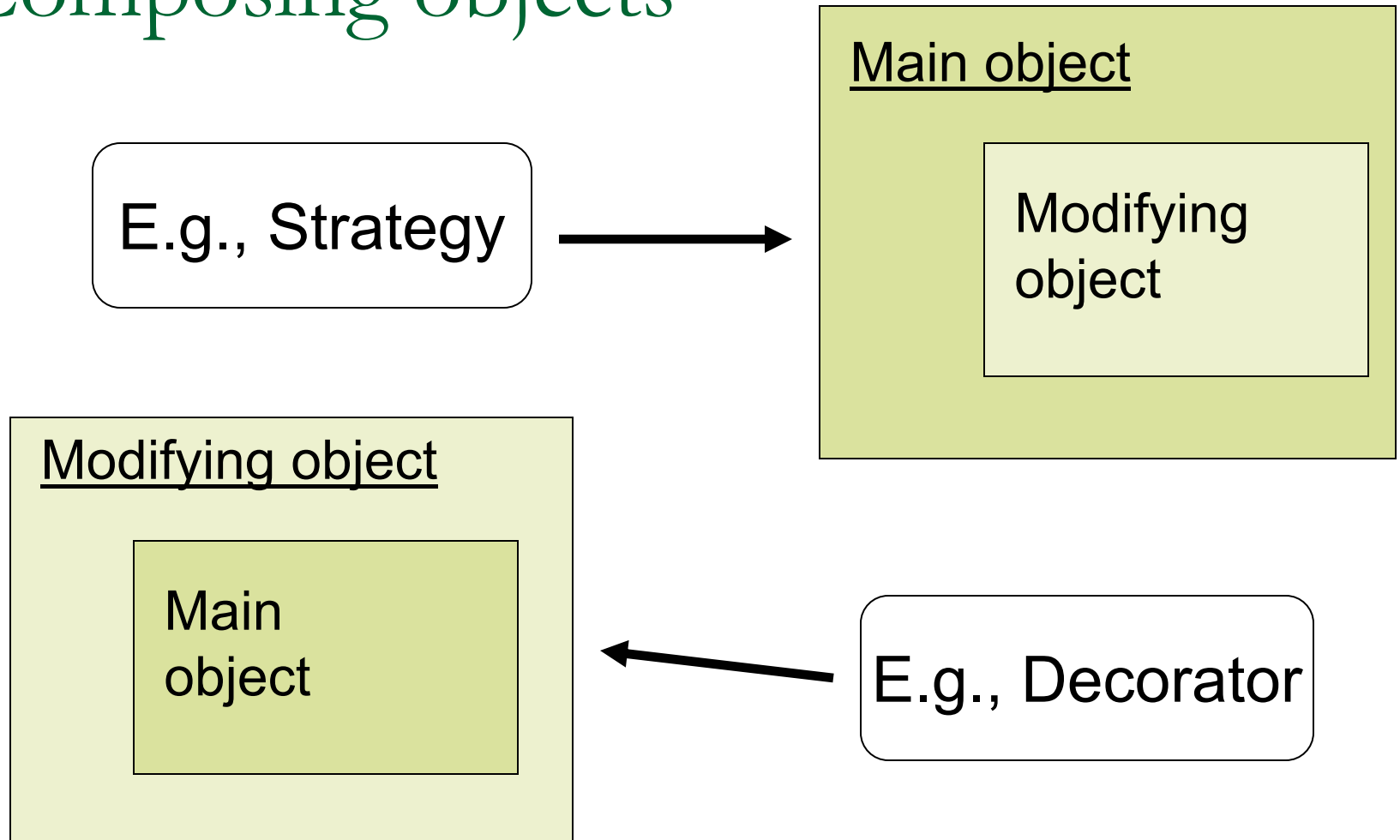
- Rather than use inheritance use composition
- Use two objects rather than one

Strategy Pattern



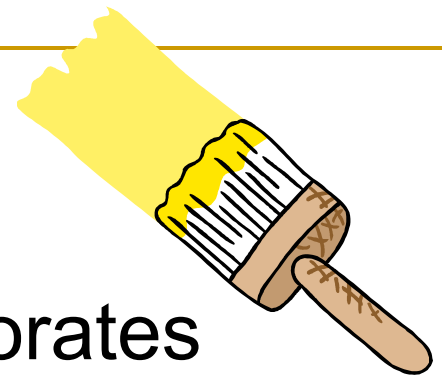
- Main object composed with some subordinate object
 - E.g., Frame has a Layout object
 - This composition is actually the **Strategy** pattern
 - Can choose an algorithm dynamically
 - Won't go into details of **Strategy**

Composing objects

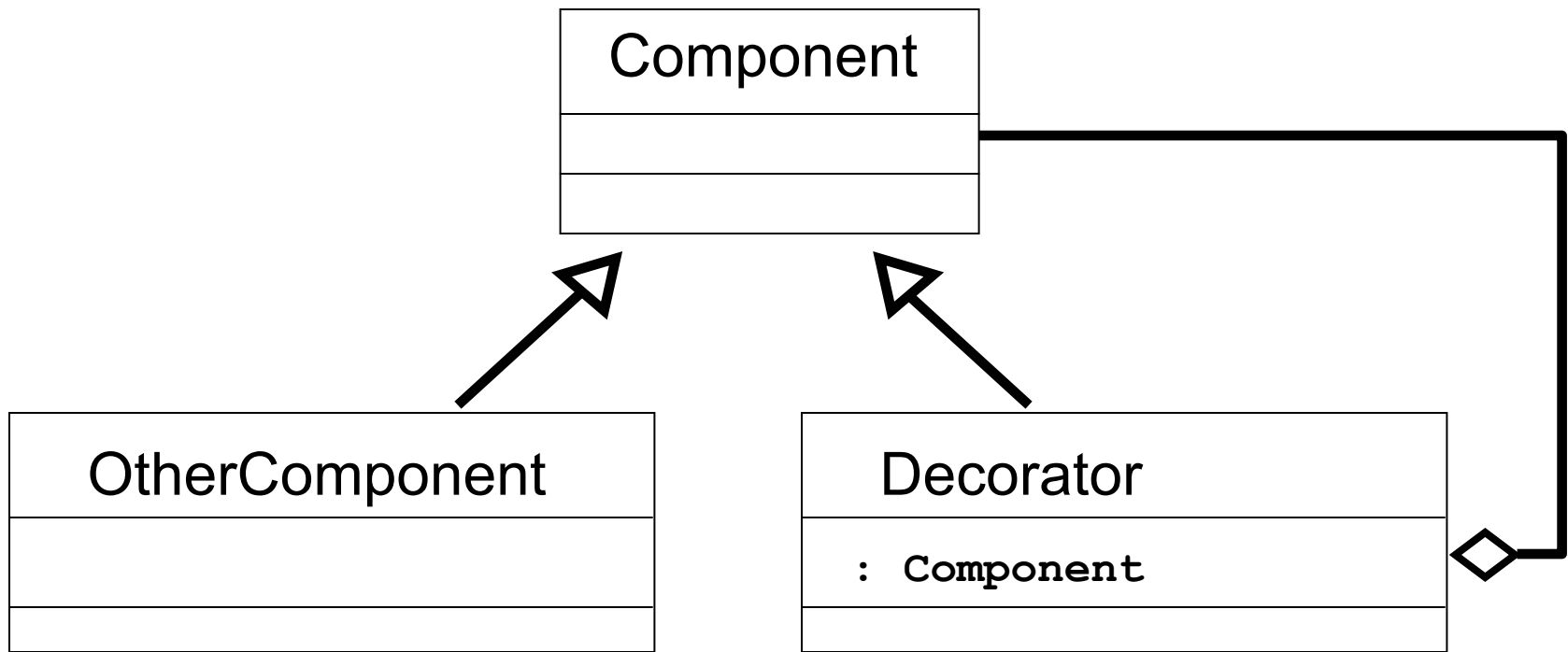


- Decorator “surrounds” the object it decorates

Decorator Pattern



- Decorator contains object it decorates
- Can substitute decorator for object it decorates



Typesetter Output:

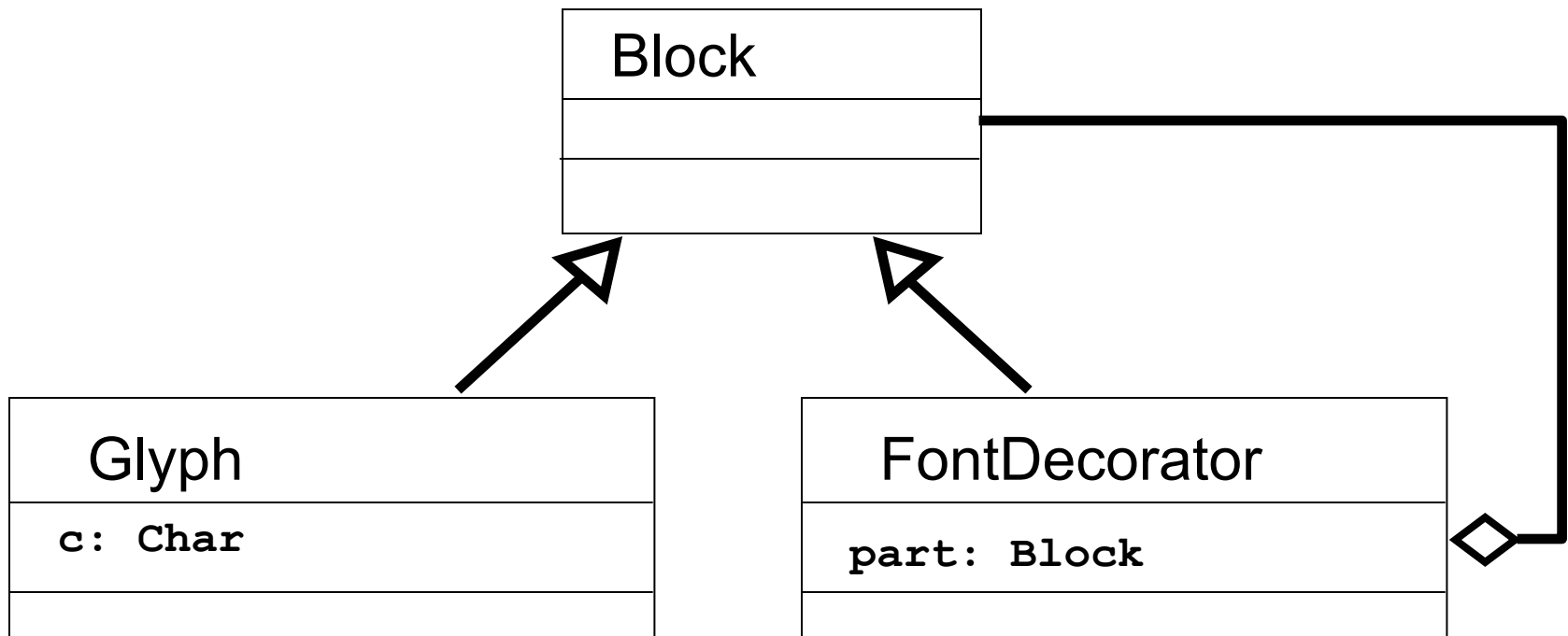
- Typesets all types of Block
 - Each Block has height and width

```
public interface Block {  
    int getHeight( );  
    int getWidth( );  
}
```

- Produce output for word processor or print driver
 - E.g., typeset letter 'H' produces output
H @ 0,0 font: italic 9pt

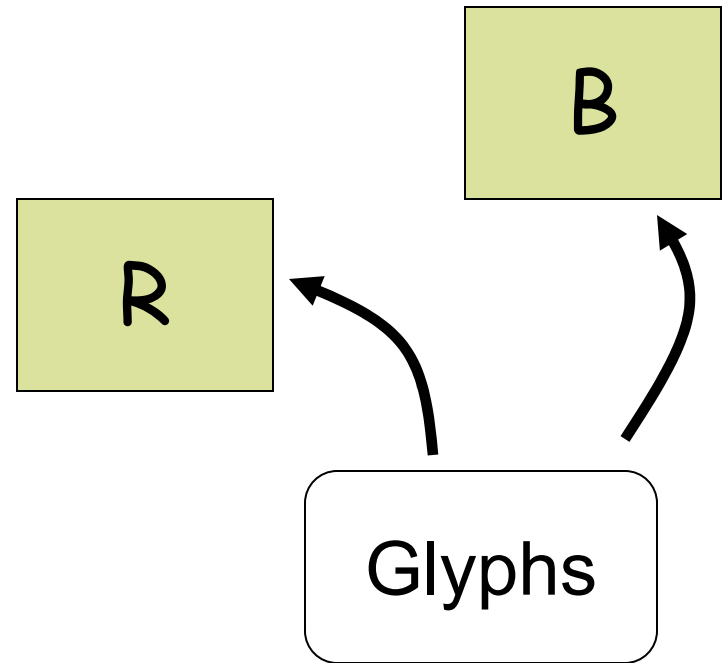
Example:

- A Glyph is a Block
- A FontDecorator is a Block



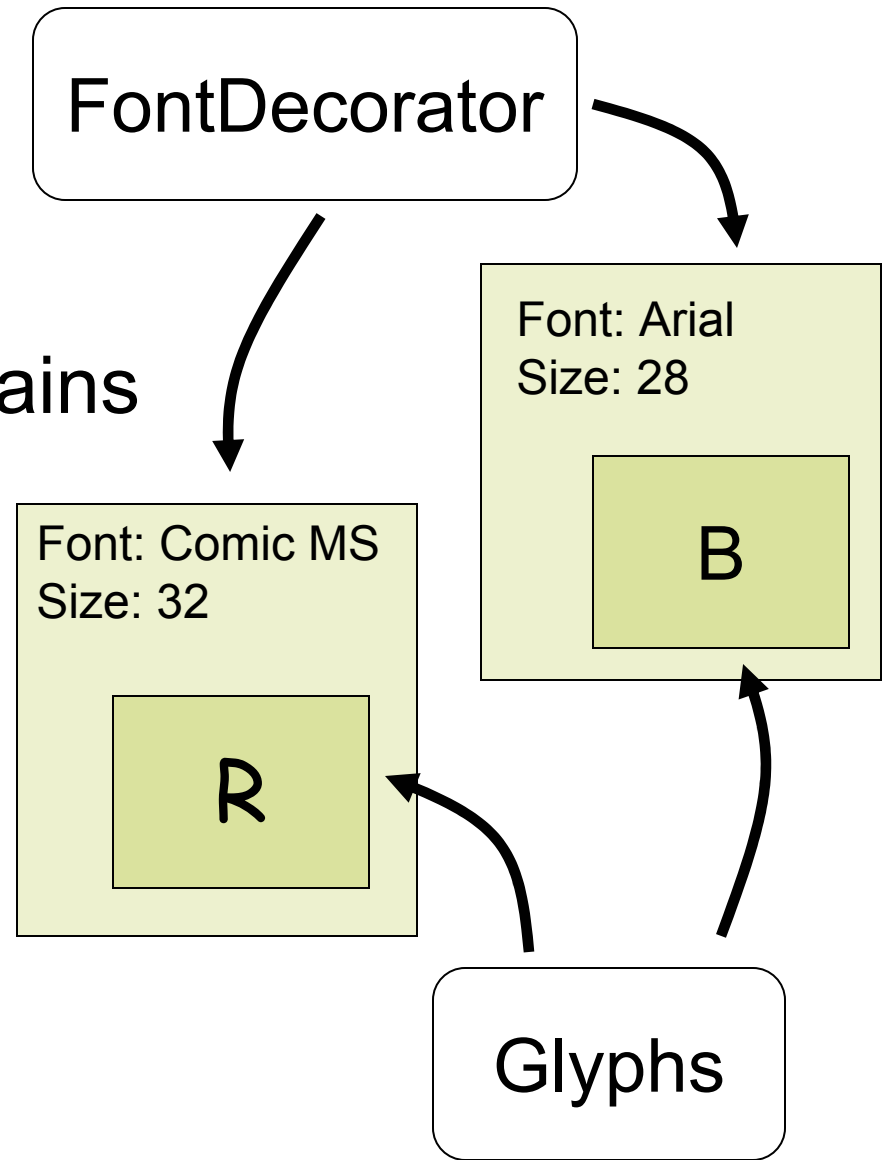
Example: Decorator

- Glyphs store single characters
- Typesetting program
 - Typesets Blocks
 - \Rightarrow Typesets Glyphs
- How to add font, size info to Glyphs?



Example:

- Use a Decorator
- `FontDecorator` contains
 - Font info
 - Size info
 - A Block
 - E.g., a Glyph

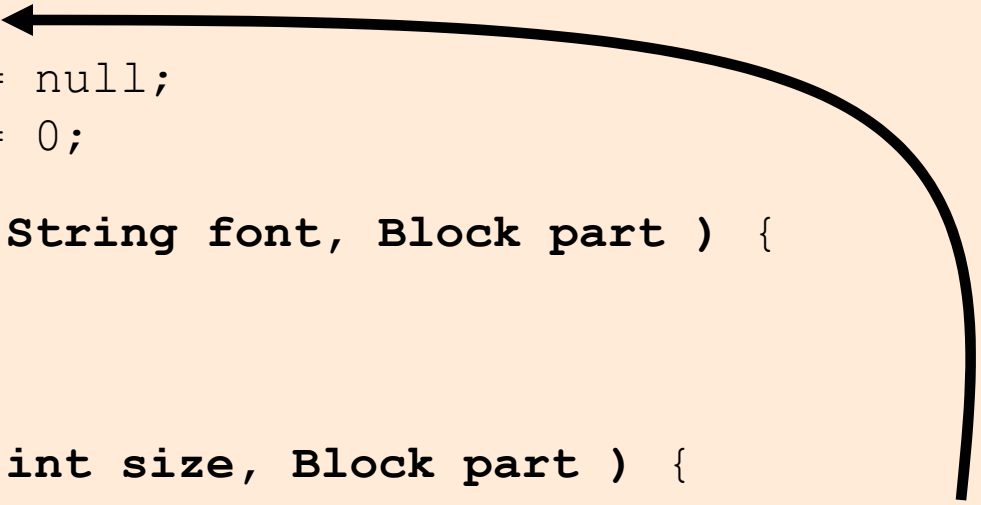


```
public class Glyph implements Block {  
    private char _c;  
  
    private Glyph( char c ) {  
        _c = c;  
    }  
  
    public int getHeight( ) { return 1; }  
    public int getWidth( ) { return 1; }  
}
```


- **Glyph implements Block and stores a char**

```
public class Typesetter {
    private int _x, _y;
    private int _size;
    private String _font;
    ...
    // some code that sets up position, font and size
    ...
    public void typeset( Glyph g ) {
        System.out.println(g.getChar( ) + " @ " + _x +
            "," + _y + " font: " + _font + " " + _size + "pt"
        );
    }
}
```


- Typesetter **class** does the formatting
- **E.g.**, typeset(Glyph)
H @ 0,0 font: italic 9pt

```
public class FontDecorator implements Block {  
    private Block _part;   
    private String _font = null;  
    private int _size = 0;  
  
    public FontDecorator( String font, Block part ) {  
        _part = part;  
        _font = font;  
    }  
    public FontDecorator( int size, Block part ) {  
        _part = part;  
        _size = size;  
    }  
  
    public String getFont() { return _font; }  
    public int getSize() { return _size; }  
    public Block getPart() { return _part; }  
    public int getWidth() { return _part.getWidth(); }  
    public int getHeight() { return _part.getHeight(); }  
}
```

Info from
contained class



```
public class Typesetter {  
    ...  
    public void typeset( Glyph g ) {...}  
  
    public void typeset( FontDecorator d ) {  
        _size = d.getSize( );  
        _font = d.getFont( );  
  
        typeset(d.getPart() );  
    }  
}
```



Recursively call `typeset()` ¹

- Typesetting a `fontDecorator` extracts `fontDecorator` info
 - Then recursively calls `typeset()` with contained info
 - Contained info could be another `fontDecorator`!

[1] Psuedo-code only. Type error here since `getPart()` returns a `Block`.

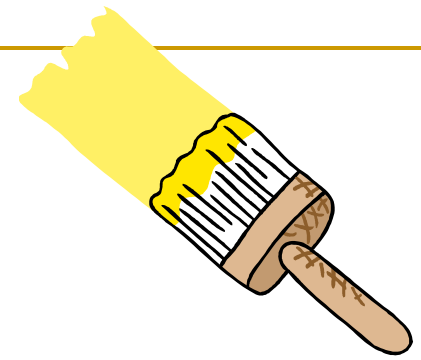
Usage:

```
public static void main( String[] args ) {
    Block b = new FontDecorator( "italic",
        new Glyph( 'H' ) ) );
}
```

- But `fontDecorator` is itself a `Block`
 - Can nest `fontDecorators`!

```
public static void main( String[] args ) {
    Block b = new FontDecorator( "italic",
        new FontDecorator( 9,
            new Glyph( 'H' ) ) );
}
```

Decorator: Consequences



- More flexible than static inheritance
 - E.g., decorate single objects not entire class
- Avoid feature-laden classes high in class hierarchy
 - Don't have to implement all foreseeable functionality up front
- Disadvantage: lots of little classes
 - Hard to understand/debug!

Visitor

- Complex pattern
 - Challenge!
 - Problem: want to group related functions
 - Don't want to search through every class to understand one function
 - E.g., to understand typesetting, have to look through every `Block` class.
 - Problem: we want to operate on a diverse set of classes
 - Each class has a different interface
-

Visitor: Solution

- Essentially use a glorified (but elegant) switch statement....

```
switch ClassType {  
  case Class1:  
    Do something...  
  case Class2:  
    Do something else...  
  case Class3:  
    Do something more...  
  ...  
}
```

Visitor: Solution

- Except use method overloading...

```
visit(Class1) {  
    Do something...  
}  
visit(Class2) {  
    Do something else...  
}  
visit(Class3) {  
    Do something more...  
}  
...
```

- ...and some other trickery

Visitor: Solution

- A visitor has functionality
 - E.g., has a `visit()` method say
- Visitor “visits” those objects that will “accept”
- Accepting object utilises functionality offered passing reference to itself
 - E.g., Accepting object calls
`visitor.visit(this)`

Visiting Object:cat

Accepting
Object:bear



```
bear.accept(cat)
```



```
accept(visitor) {  
  visitor.visit(bear)  
}
```



Typsetter Example

- Typsetter offers functionality
 - `typeset()` method
 - Rename it `visit()`
- Let Typsetter implement some Visitor interface
- Glyphs wants to accept that functionality
 - Willing to accept a “visit” from the Typsetter
 - All accepting classes have following code:

```
public class Glyph {  
    ...  
    public void accept( Visitor visitor ) {  
        visitor.visit( this );  
    }  
}
```

Visiting Object:Typesetter

Accepting
Object:Glyph

glyph:
Glyph

accept(Visitor)

glyph.accept(typesetter)

typesetter:
Typesetter

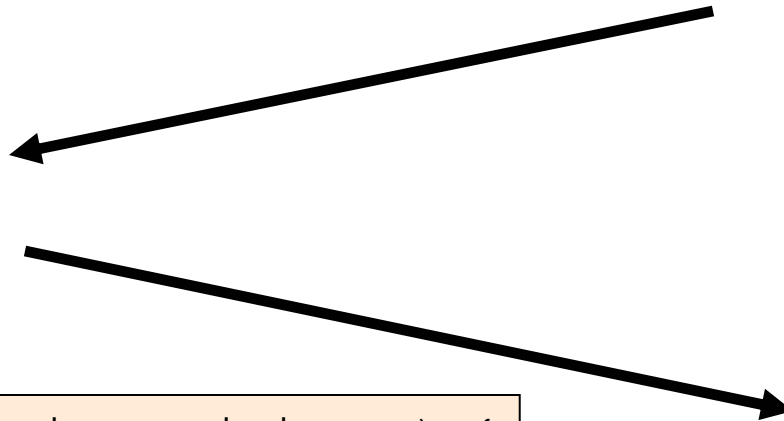
Visit(Block)

```
accept( Visitor visitor ) {  
    visitor.visit( this );  
}
```

typesetter:
Typesetter

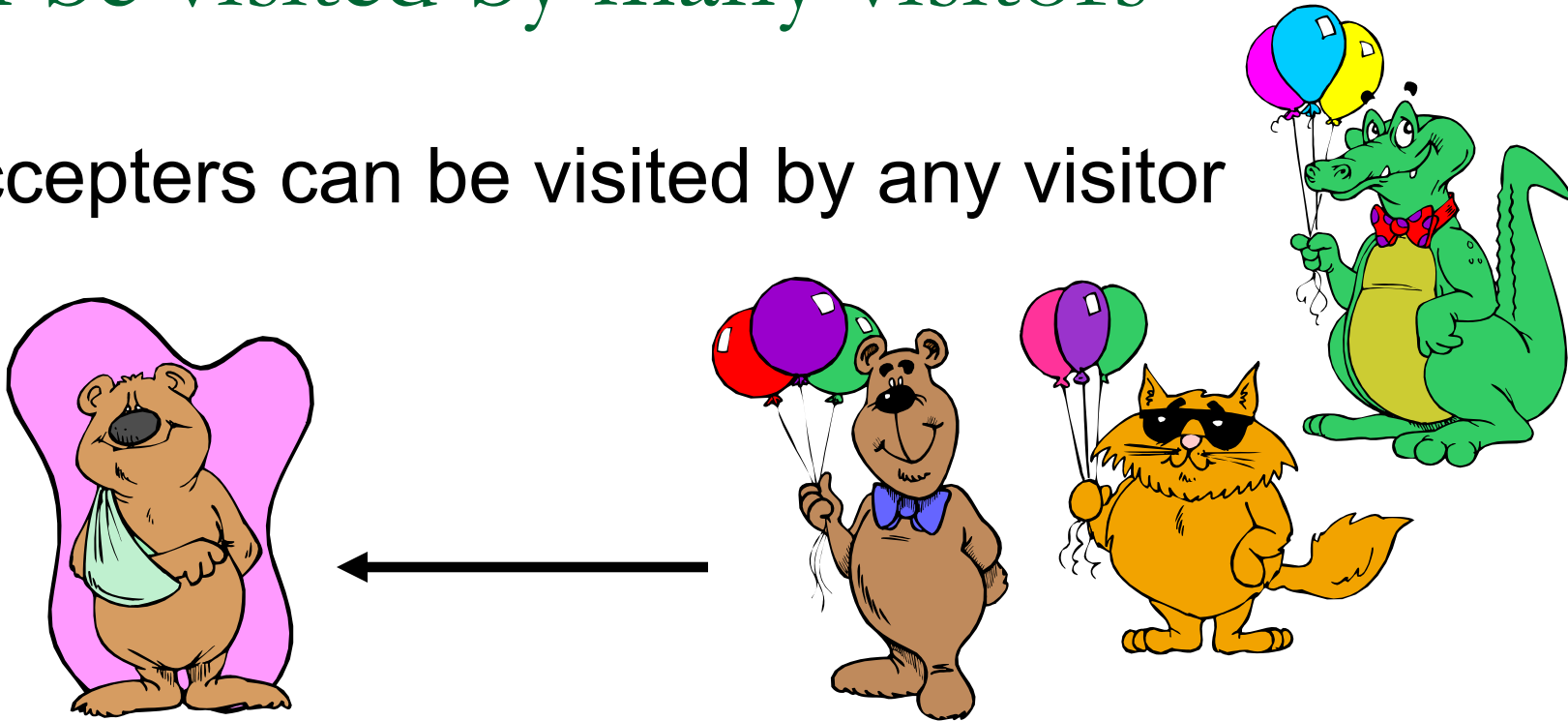
Visit(Block)

Do typesetting



Can be visited by many visitors

- Accepters can be visited by any visitor

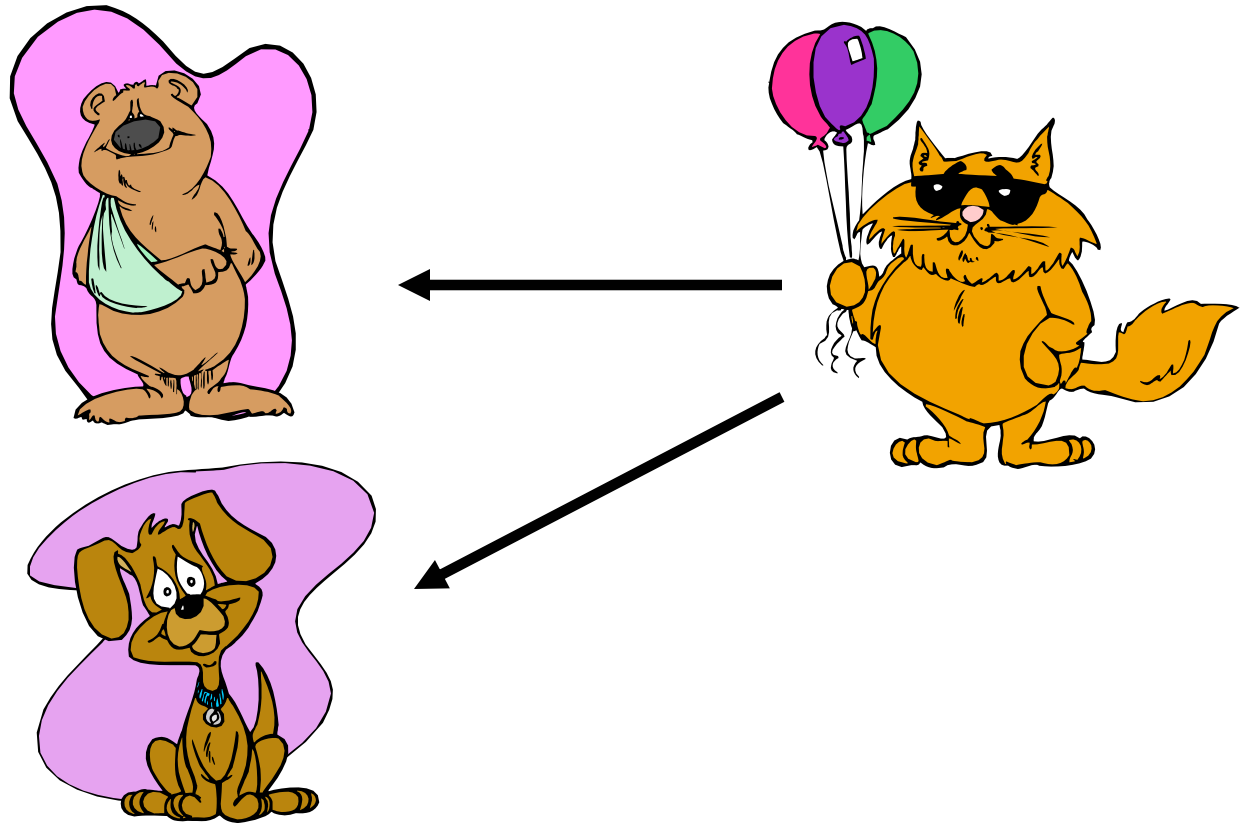


- Acceptor must implement `accept()` method

```
public void accept( Visitor visitor ) {  
    visitor.visit( this );  
}
```

Visitor can make multiple visits

- Visitors can visit any accepting class



Visitor can make multiple visits

- Visitors must simply be able to handle any accepting class
 - E.g., for the typesetter example `Visitor` must handle any type of `Block`
 - I.e., if there are four types of `Block` have:

```
public interface Visitor {  
    void visit( Glyph g );  
    void visit( Horizontal h );  
    void visit( Vertical v );  
    void visit( FontDecorator d );  
}
```

Consequences



- Easy to add new operations
- Gathers related operations
- Disadvantage: Visited (accepting) classes are hard to subclass
 - ❑ Visitor must deal with subclasses correctly
- Disadvantage: Breaks encapsulation
 - ❑ E.g., `Glyph` no longer manages its typesetting
 - ❑ Accepting class usually needs to make internals visible to visitor. Breaks Info Hiding!

Summary

Remember: Decorator is simply the
name of the pattern.
It does *not* imply use of graphics.

- Decorator

- Add functionality to individual object
- “Surround” object with decorator

- Visitor

- Gather together functionality

