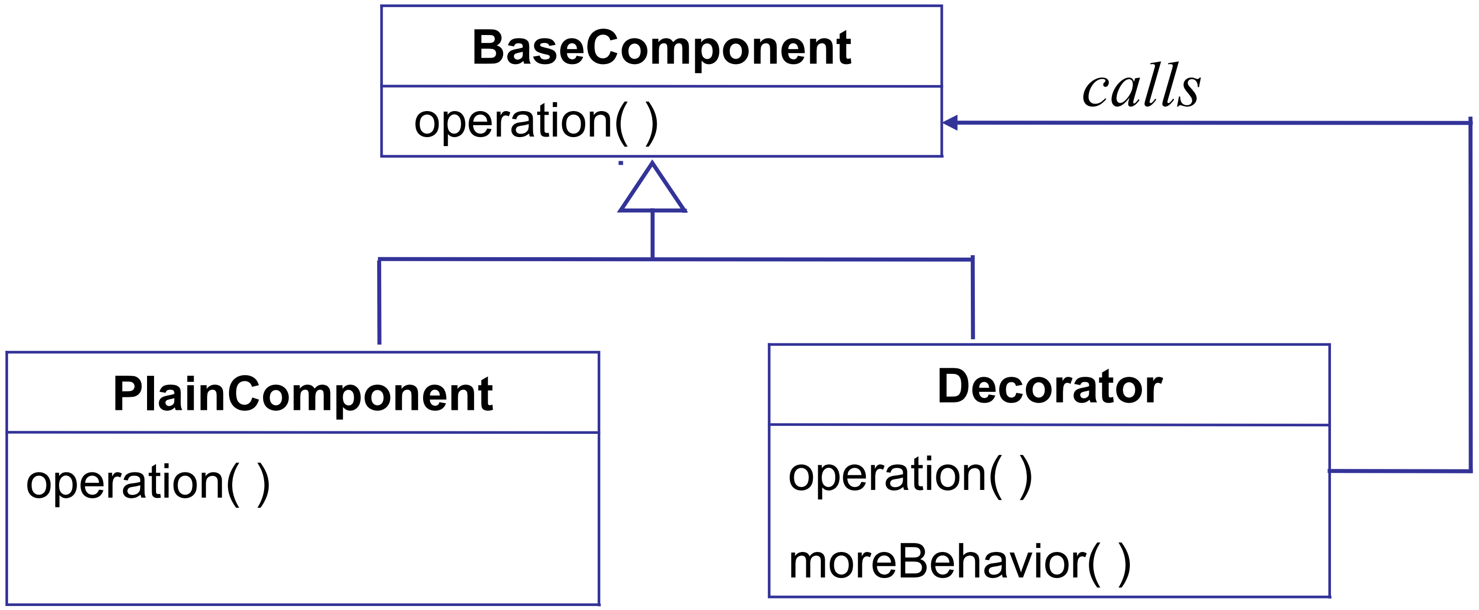
Decorator Pattern

Context: We want to *enhance* the behavior of a class, without making the class more complicated.

The *enhanced* class can be used the same as the base class.

Solution: A base class or interface defines the required behavior. Create a *decorator* that implements the base interface and wraps an instance of the plain class, "decorating" its behavior.

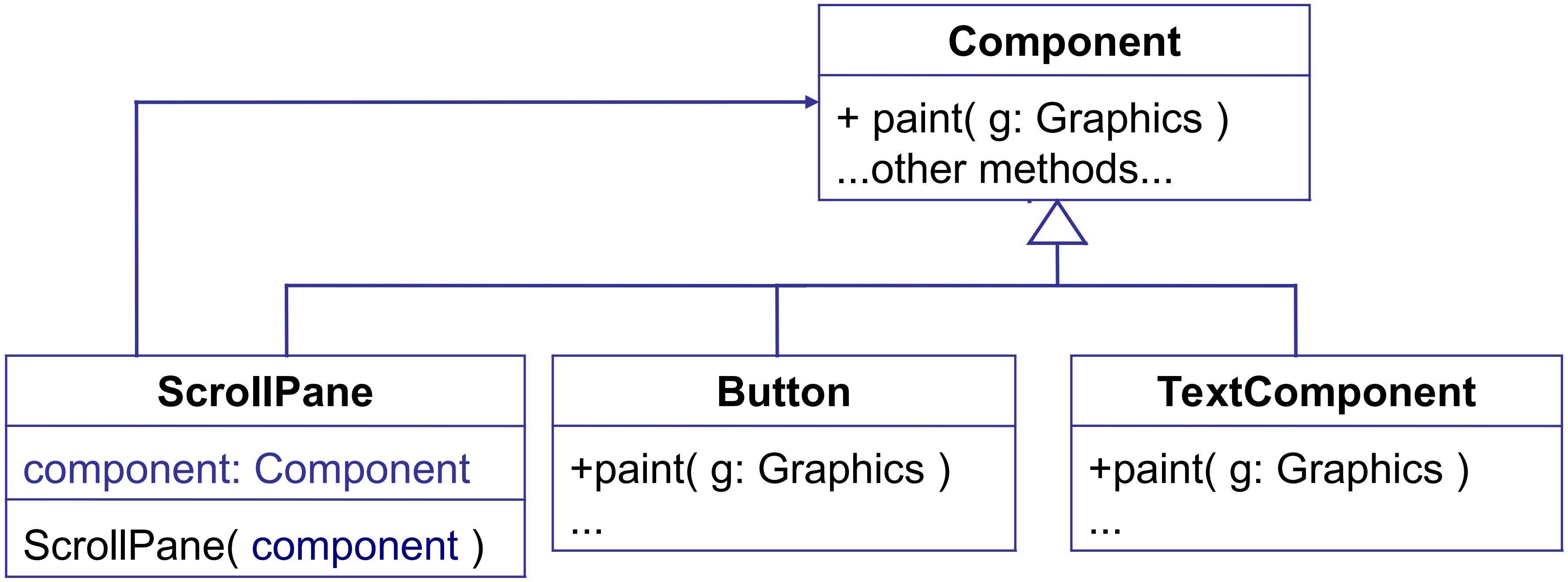


Decorator Example

Context: We want to add Scroll Bars to different graphical components. We don't want duplicate code for Scroll Bars

Solution: Component is the base class for all components. ScrollPane "wraps" any component and adds scroll bars to it.

We can "wrap" any component with a Scroll Pane and the component behaves the same, but has scroll bars



Decorator Example

Purpose: create a TextArea with scrollbars so that text will scroll when larger than the viewport.

```java
// a TextArea with 5 rows and 40 columns
JTextArea textArea = new JTextArea( 5, 40 );

// decorate with JScrollPane to add scrollbars
JScrollPane pane = new JScrollPane( textArea );
pane.setVerticalScrollBarPolicy(
   JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED );

// Add the decorated component to the window
// *instead of* the original textArea
window.add( pane );
```

Advantage of Using Decorators (1)

- We can write the decorator behavior one time and apply it to many different kinds of objects.

Example: a JScrollPane can be applied to any kind of Component, even buttons!

Advantage of Using Decorators (2)

- Improves the *cohesion* of objects, by not adding extra responsibility that isn't part of the object's main purpose.

Example: the purpose of a TextArea is to display text!

Not to manage scroll bars.

Advantage of Using Decorators (3)

- New decorators can be added in the future, *extending* the behavior of the class.

Example: a *zoom decorator* to zoom a component.

Open-Closed Principle

A class should be open for extension but closed for modification.

Disadvantage of Decorators

Lots of pass-through methods

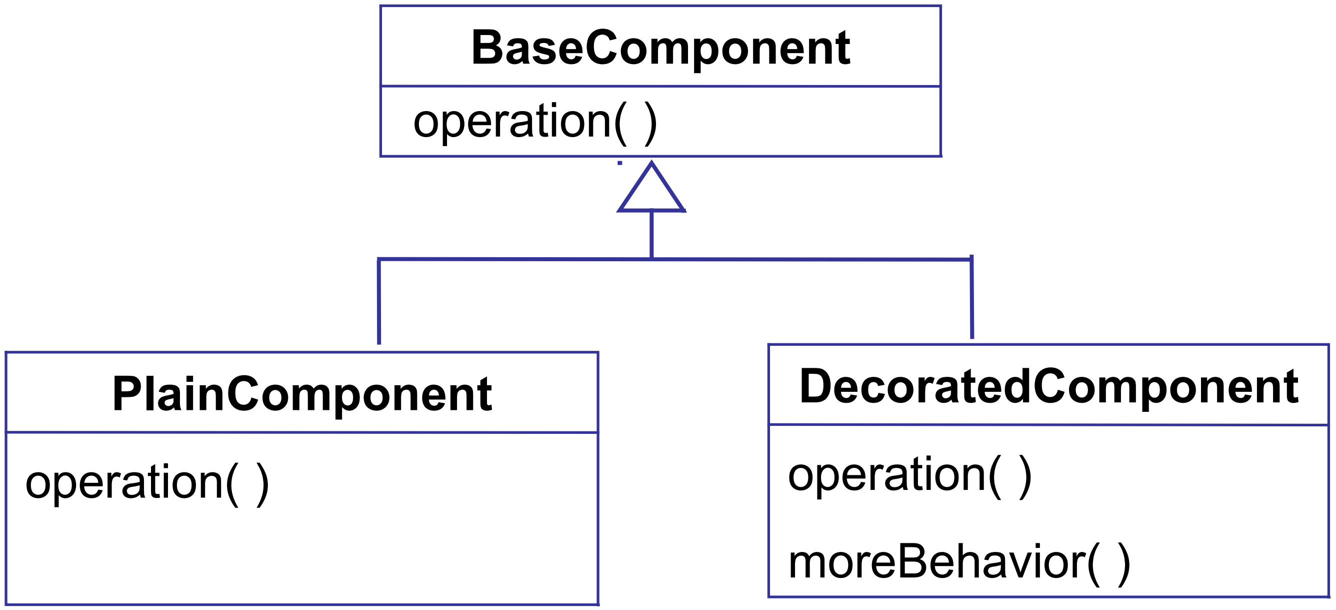
Any method the decorator doesn't "decorate" itself, it must pass to the decorated object.

Class Decorator?

Usually a Decorator encapsulates another instance of the base type, and calls its methods. This is composition.

But, if you only want to decorate a single base type you could define the decorator as a subclass that directly uses the superclass.

That means you create a Decorator object instead of creating a base type object.

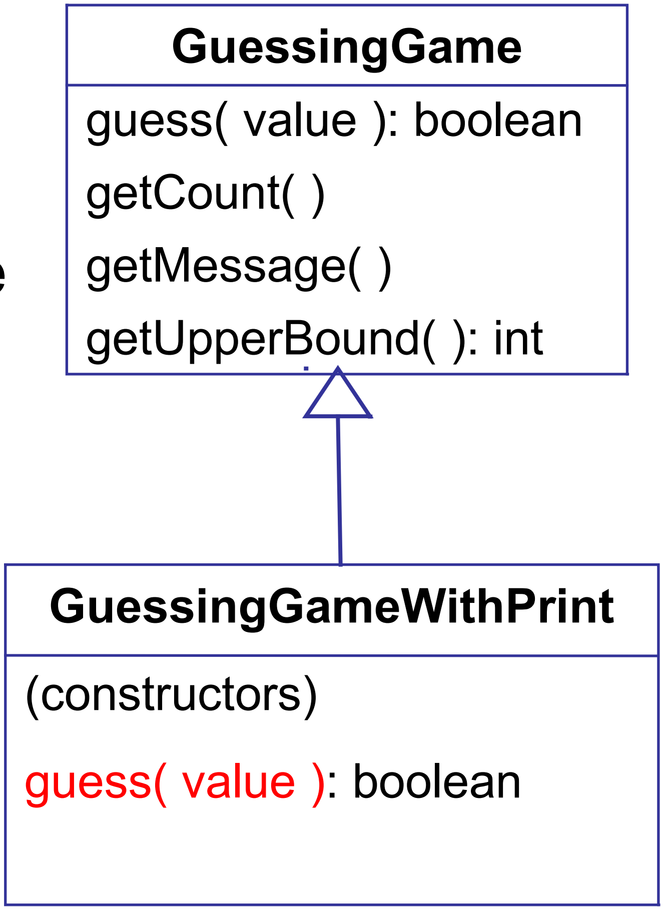


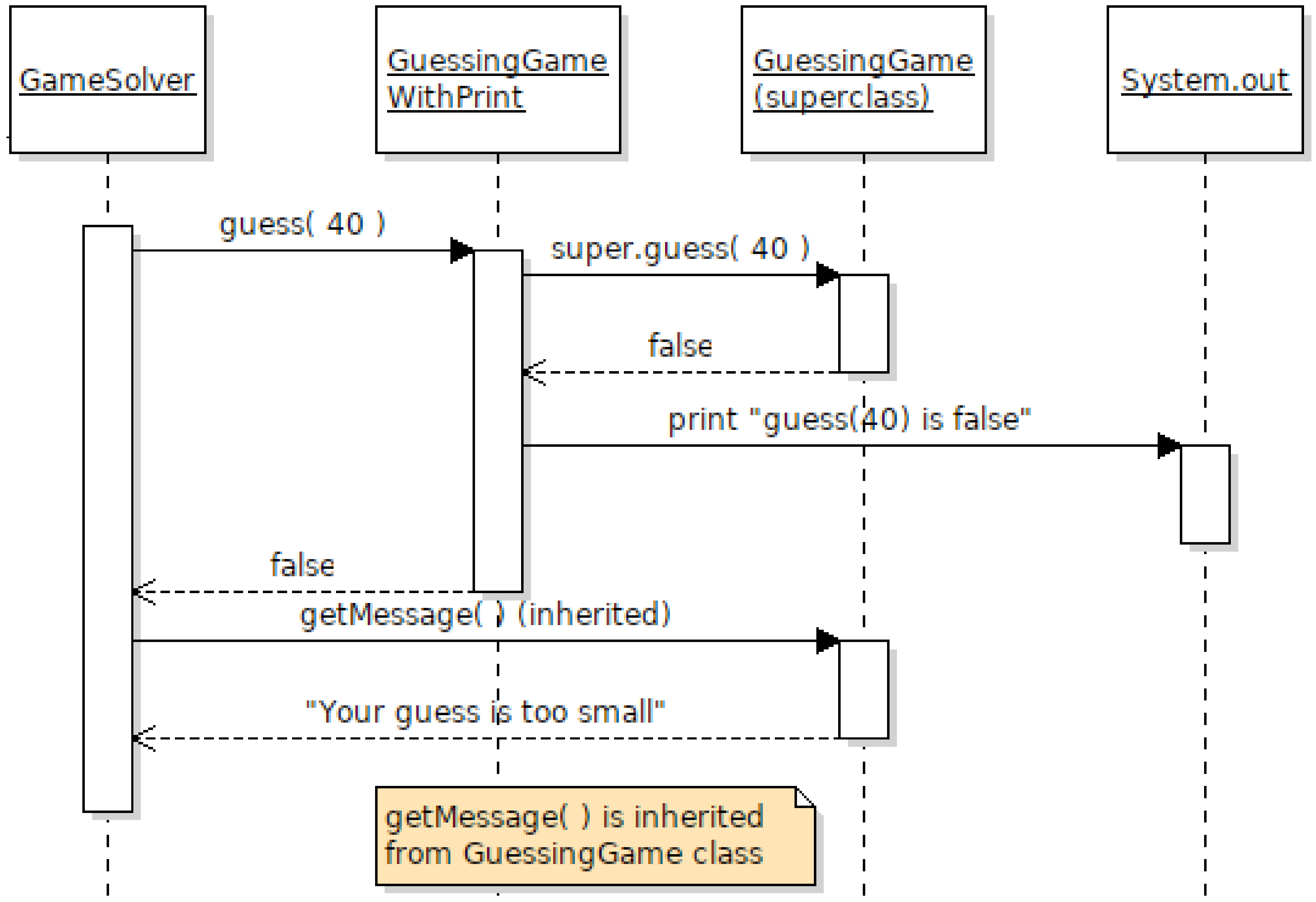
GuessingGame Decorator

A GuessingGameWithPrint class that *extends* the Guessing Game class.

GuessingGameWithPrint overrides the guess( ) method to print the guess, call the superclass guess(), and return value (true or false).

Other methods it simply *inherits* from GuessingGame.




GameSolver
GuessingGame WithPrint
GuessingGame (superclass)
System.out
guess(40)
super.guess(40)
false
print "guess(40) is false"
false
getMessage( ) (inherited)
"Your guess is too small"
getMessage( ) is inherited from GuessingGame class

Example Code

```
class GuessingGameWithPrint
              extends GuessingGame {
    // must provide all required constructors
    public GuessingGameWithPrint() {
        super( );
    }
    public GuessingGameWithPrint(int bound) {
        super(bound);
    }
    public boolean guess(int value) {
        boolean result = super.guess(value);
        System.out.printf("guess(%d) is %b\n",
              value, return);
        return result;
    }
```

System.out.printf( )

`printf` prints a *formatted string* with data.

Syntax:

```
printf( format, arg1, arg2, ... )
```

Example:

```
printf( "Hello %s, the day is %d\n", "Nok", 22 );
Hello Nok, the day is 22
```

See:

https://dzone.com/articles/java-string-format-examples

Python Function Decorator

Context: We want to see each time a function is called and what the function returns.

Forces: We don't want to modify the code (add "print" statements), and a debugger is too cumbersome & slow.

Solution: Wrap the function in *another function* that prints each time it is called.

```
def decorate(fun):
    """fun is a function to decorate."""
    def new_fun(*args, **kwargs):
        s = ", ".join(str(arg) for arg in args)
        print(f"{fun.__name__}({s})")
        return fun(*args, **kwargs)
    return new_fun

f = decorate(fibonacci)    # decorate fibonacci
```