Decorator Pattern

Abram Hindle

University of Alberta
Edmonton, Canada
http://softwareprocess.es/

abram.hindle@softwareprocess.es

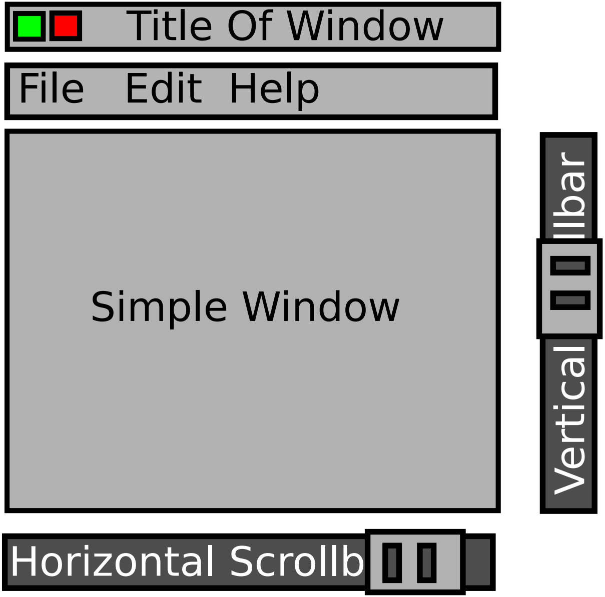
Introduction

- How do we change the behaviour of an object (an instance) rather than a class?
- How to change behaviour of a class at runtime?
- How can we separate multiple responsibilities?

Example: Window Decorations

- Imagine we have a window. It can have:
  - Titlebar
  - Menubar
  - Vertical Scrollbar
  - Horizontal Scrollbar

Window Decorators



Title Of Window

File Edit Help

Simple Window

Vertical Scrollbar

Horizontal Scrollb

Example: Window Decorations

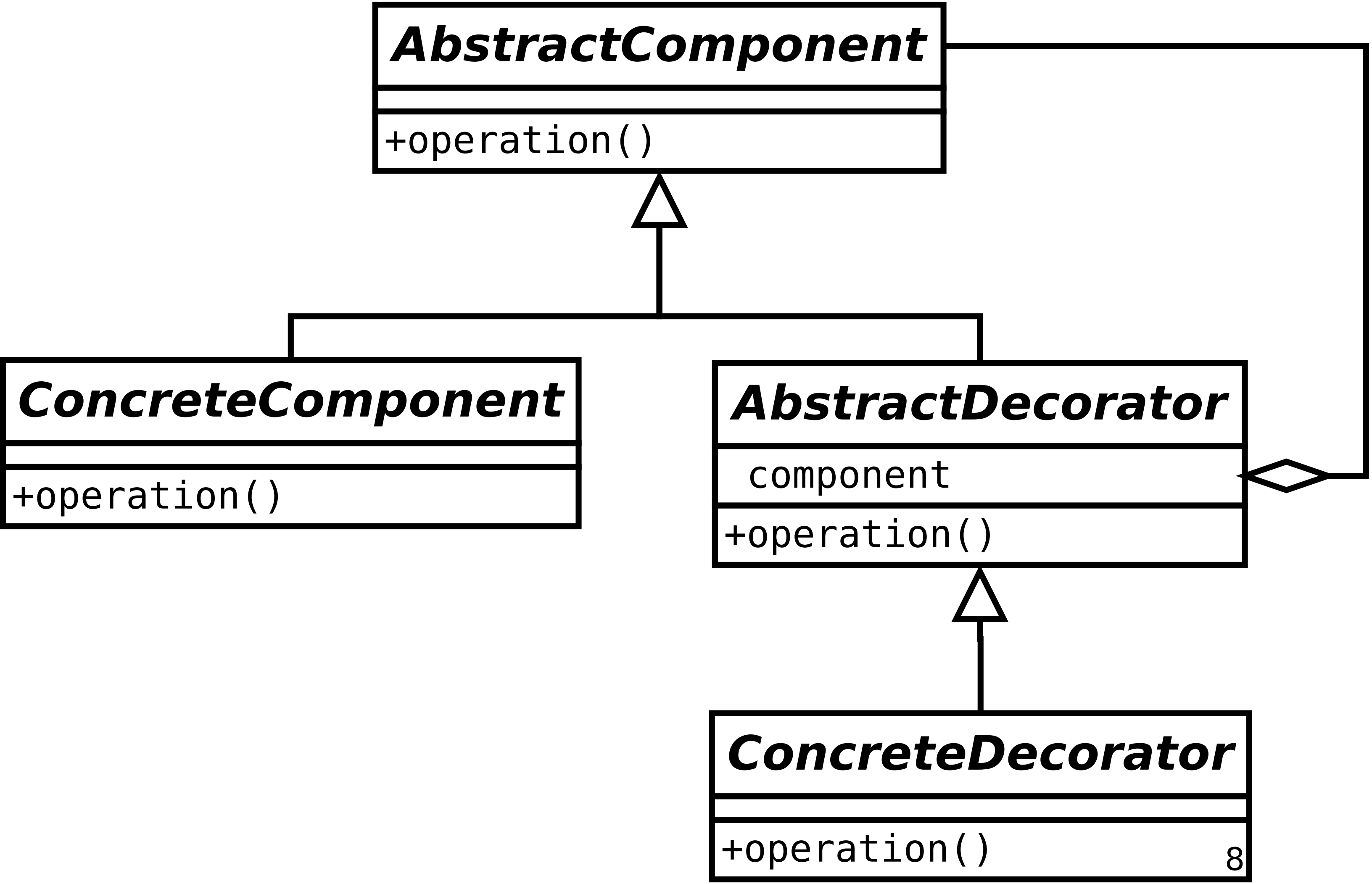
- Should we implement 1 class which has all 4 responsibilities?
  - What if we don’t want them all?
  - Should we couple a class with unrelated responsibility?
  - Should a window implement all of the widgets inside of it?

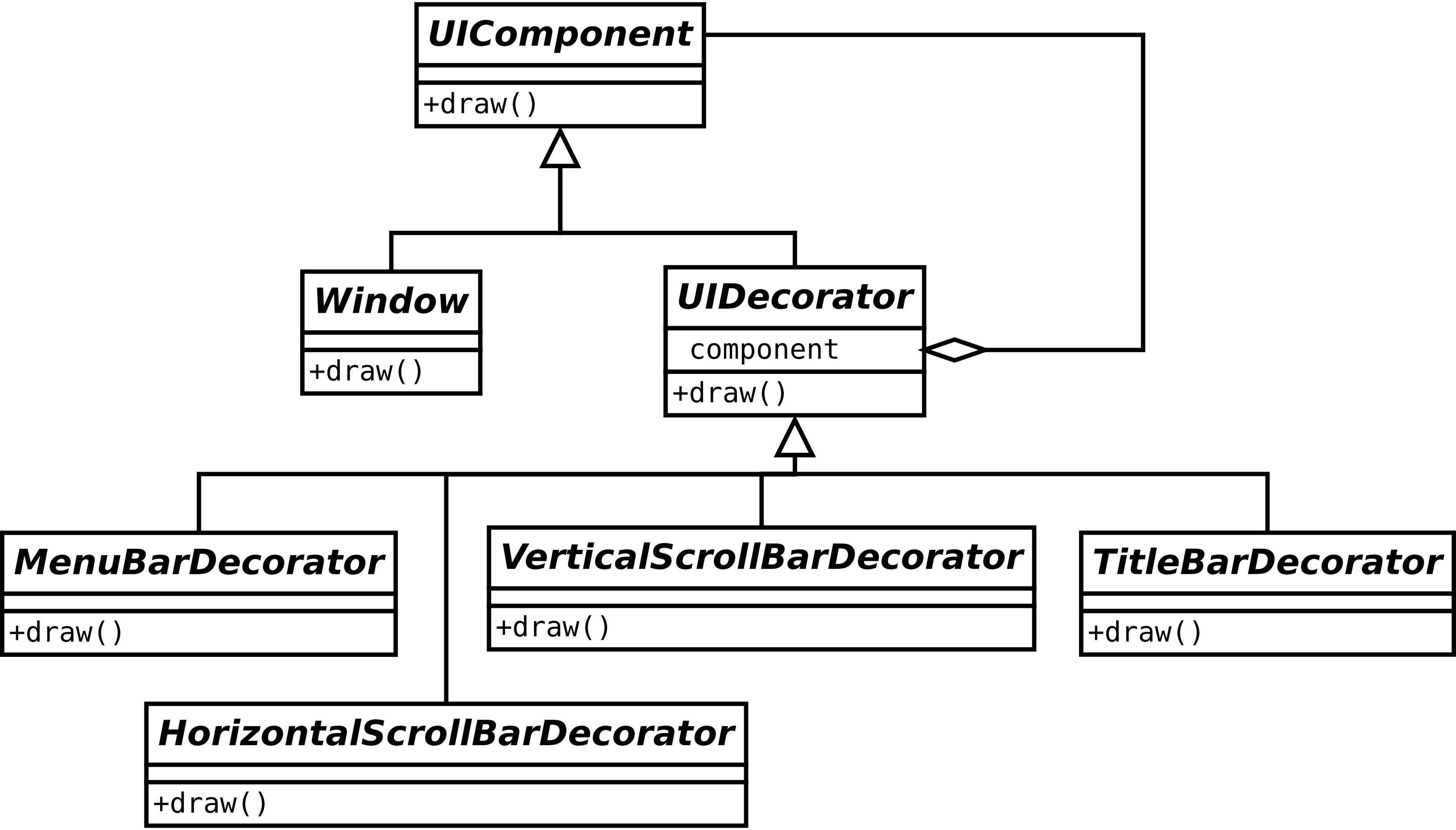
Example: Window Decorations

- Should we implement 16 different combinations of windows?
  - TitleBarWindow
  - TitleBarMenubarWindow
  - TitleBarMenubarVerticalScrollbarWindow
  - TitleBarMenubarVerticalScrollbarHorizontalScrollbarWindow
- If we have these classes, we can’t change behaviour at runtime!

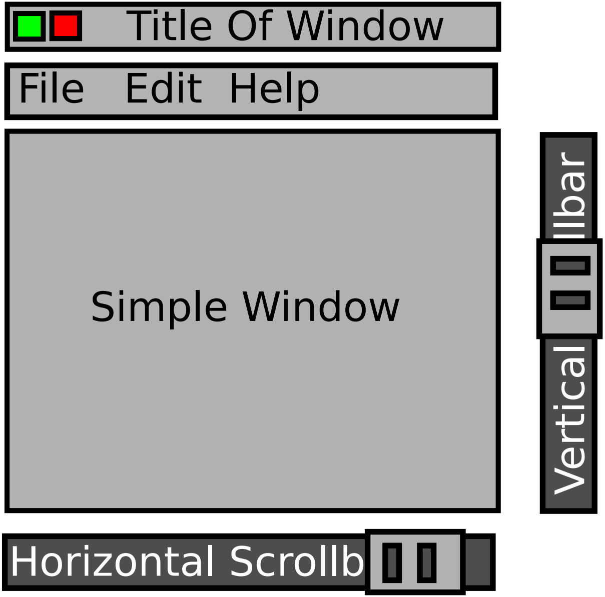
What is a potential solution?

- The decorator pattern!
  - Let’s wrap our Window with decorators who fulfill these responsibilities
  - Each component can be responsible for drawing itself.
  - Separate responsibilities to responsible component.
  - Avoid combinatorial explosion of classes


AbstractComponent
+operation()
ConcreteComponent
+operation()
AbstractDecorator
component
+operation()
ConcreteDecorator
+operation()


UIComponent
+draw()
Window
+draw()
UIDecorator
component
+draw()
MenuBarDecorator
+draw()
VerticalScrollBarDecorator
+draw()
TitleBarDecorator
+draw()
HorizontalScrollBarDecorator
+draw()

Window Decorators



Title Of Window

File Edit Help

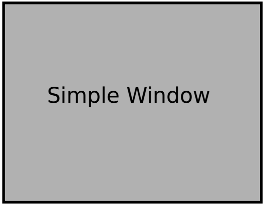
Simple Window

Vertical llbar

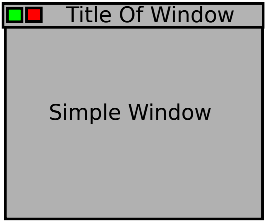
Horizontal Scrollb

Window Decorator Example

```
UIComponent w, tb, mtb, hsbmtb;
w = new Window();
w.draw();
```



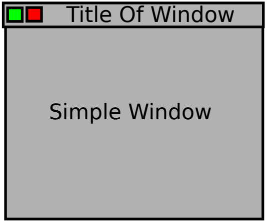
```
tb = new TitleBarDecorator( w );
tb.draw();
```



Window Decorator Example

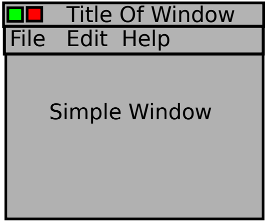
```
tb = new TitleBarDecorator( w );
```

```
tb.draw();
```



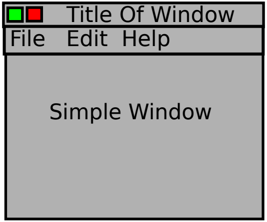
```
mtb = new MenuBarDecorator( tb );
```

```
mtb.draw();
```



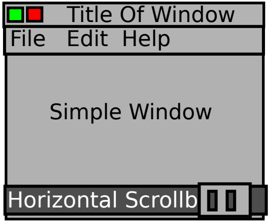
Window Decorator Example

mtb = new MenuBarDecorator(tb);



mtb.draw();

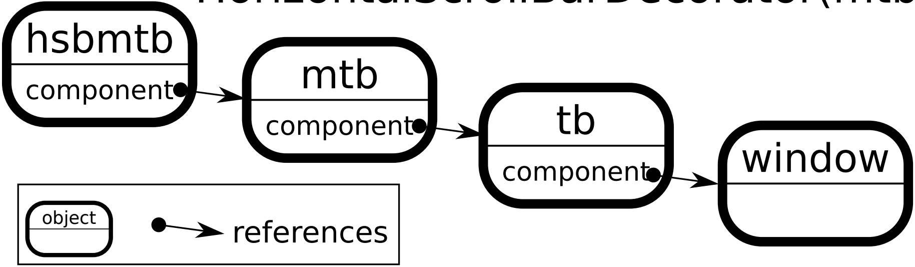
hsbmtb = new HorizontalScrollBarDecorator(mtb)

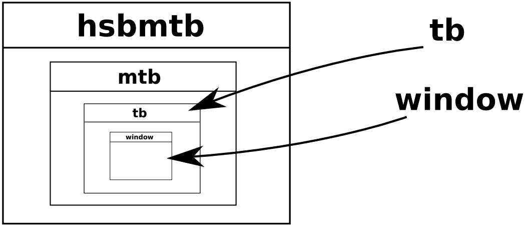


hsbmtb.draw();

Window Decorator Attributes

```
hsbmtb = new
  HorizontalScrollBarDecorator(mtb);
```

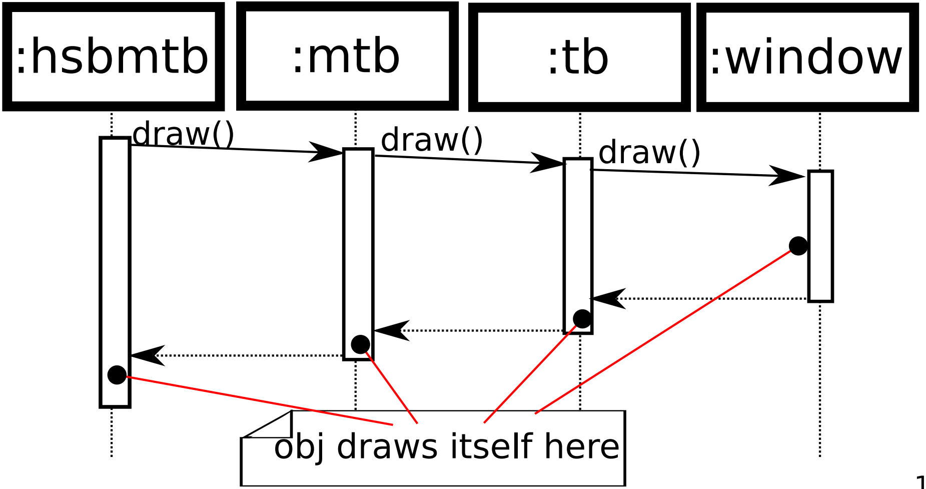




2 diagrams of the chain of references

Window Decorator Calls

```
hsbmtb = new
  HorizontalScrollBarDecorator(mtb);
```



Window Decorator Code

- Example Window Decorator Code
  - Note that the drawing routines are simulated
  - Also observe how `draw()` is implemented

WindowDecoratorDriver.java

```
interface UIComponent {
    public void draw();
}
class Window implements UIComponent {
    public void draw() {
        /* Draw Window */
        System.err.println("\nDraw␣Window");
    }
}
abstract class UIDecorator implements
     UIComponent {
    UIComponent component;
}
class TitleBarDecorator extends UIDecorator
     {
    TitleBarDecorator(UIComponent c) {
        component = c;
    }
    public void draw() {
        component.draw();
        /* draw title bar here */
        System.err.println("Draw␣TitleBar");
    }
}
class MenuBarDecorator extends UIDecorator {
    MenuBarDecorator(UIComponent c) {
         component = c; }
    public void draw() {
        component.draw();
        /* draw menu bar here */
        System.err.println("Draw␣MenuBar");
    }
}
```

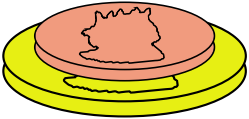
```
class HorizontalScrollBarDecorator extends
     UIDecorator {

    HorizontalScrollBarDecorator(UIComponent
           c) {
        component = c;
    }
    public void draw() {
        component.draw();
        /* draw HScroll bar here */
        System.err.println("Draw␣Horizontal␣
              Scroll␣Bar");
    }
}
```

```
public class WindowDecoratorDriver {
    public static void main(String [] argv)
         {
        UIComponent w, tb, mtb, hsbmtb;
        w = new Window();
        w.draw();
        tb = new TitleBarDecorator( w );
        tb.draw();
        mtb = new MenuBarDecorator( tb );
        mtb.draw();
        hsbmtb = new
             HorizontalScrollBarDecorator(
             mtb );
        hsbmtb.draw();
    }
}
```

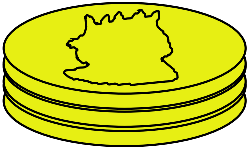
WindowDecoratorDriver.java.txt

```
Draw Window

Draw Window
Draw TitleBar

Draw Window
Draw TitleBar
Draw MenuBar

Draw Window
Draw TitleBar
Draw MenuBar
Draw Horizontal Scroll Bar
```

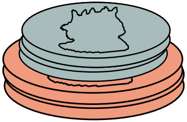
Stacked Coins Example



101 cents



300 cents



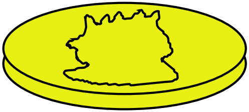
22 cents

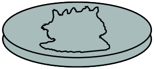
How do we represent these coins stacks?

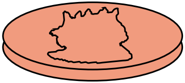
class PennyLoonie?

class DimeDimePennyPenny?

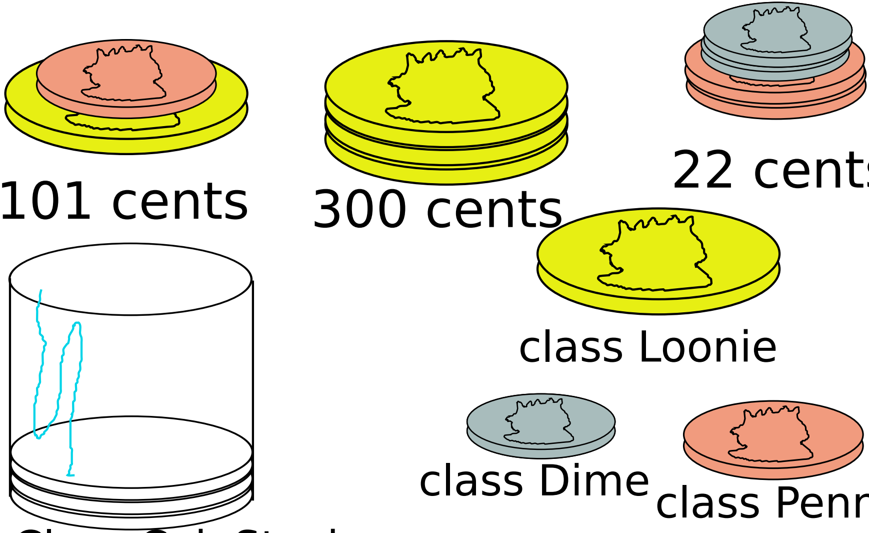
class LoonieLoonieLoonie?







Stacked Coins Example



101 cents

300 cents

22 cents

class Loonie

class Dime

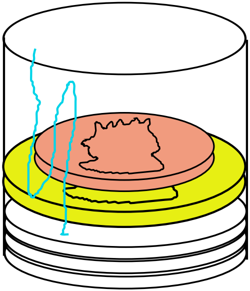
class Penny

Class CoinStack

0 cents

Stacked Coins Example

```
StackableCoin c =
  new Penny(
    new Loonie(
      new CoinStack()));
```



```
System.out.println("" +
  c.totalCentsStacked());
```

101

(0 + 100 + 1)

Stacked Coins Example

- We demonstrate a decorator for counting change
  - At runtime we can access older parts of the stack
  - We don’t need to make funny classes like LoonieAndAPenny
  - Loonies, Pennies, Dimes are all decorators for the CoinStack
  - We modify the output values
- This example is mostly for learning, not real use.

StackableCoinDriver.java

```
interface StackableCoin {
    public int totalCentsStacked();
}
// This is the concrete component
class CoinStack implements StackableCoin {
    public int totalCentsStacked() { return 0;
         }
}
// Note the lack of an Abstract Decorator
// The decorators follow
class Loonie implements StackableCoin {
    StackableCoin stack;
    Loonie(StackableCoin stack) {
        this.stack = stack;
    }
    public int totalCentsStacked() { return 100
          + stack.totalCentsStacked(); }
}
```

Decorator Pattern

```
class Dime implements StackableCoin  {
    StackableCoin stack;
    Dime(StackableCoin stack) {
        this.stack = stack;
    }
    public int totalCentsStacked() { return 10
         + stack.totalCentsStacked(); }
}
class Penny implements StackableCoin {
    StackableCoin stack;
    Penny(StackableCoin stack) {
        this.stack = stack;
    }
    public int totalCentsStacked() { return 1 +
          stack.totalCentsStacked(); }
}


public class StackableCoinDriver {
    public static void main(String args[]){
        StackableCoin empty = new CoinStack();
        System.out.println("" + empty.
             totalCentsStacked() );
        StackableCoin first = new Loonie( empty
              );
        System.out.println("" + first.
             totalCentsStacked() );
        StackableCoin second = new Penny( first
              );
        System.out.println("" + second.
             totalCentsStacked() );
        // But I can still reference first
        System.out.println("" + first.
             totalCentsStacked() );
    }
}
```

Decorator Operation Method

- Here’s a template for the operation method in the decorators

OperationExample.java

```
void operation() {
    // put code before the operation here
    ...
    // Call the wrapped component
    component.operation();
    // put code that follows the operation here
    ...
}
```

Decorator Debugging Example

- Maybe you want to add runtime debugging functionality?
- Large systems tend to produce log messages for status and debugging
  - Maybe you don’t want to couple your class with the logger, a logging decorator would make sense.

DebugExample.java

```
void operation() {
    System.err.println("Calling␣operation␣on
         ␣component:␣" + component.toString
         ());
    component.operation();
    System.err.println("Returned␣from␣
         calling␣operation␣on␣component:␣" +
          component.toString());
    Logger.log("Successfully␣called␣
         operation␣from␣LoggingDecorator");
}
```

debuglog.txt

```
1299649240: Successfully called operation
     from LoggingDecorator
1299650004: Successfully called operation
     from LoggingDecorator
1299650407: Uncaught Exception
1299651121: Successfully called operation
     from LoggingDecorator
1299651852: Successfully called operation
     from LoggingDecorator
```

Where will you see this pattern?

- User Interfaces
- Filters and Input/Output chains
- Dataflow
  - Audio (PD, CSound, Max, Reaktor)
  - Text (UNIX)
- Indirection
- Functional Programming

Decorator Fact Sheet

- Structural
- Intent: attach responsibility dynamically
- Also Known as: Wrapper
- Applicability: add or remove responsibilities without subclassing
- Participants: Component, ConcreteComponent, Decorator, ConcreteDecorator
- Uses: I/O Streams, Widgets, Buffers
- Related patterns: Adapter, Composite, Strategy

Conclusions

- Decorator pattern is used to decorate an object at runtime with extra responsibilities provided by the decorators.
- Decorator patterns work by wrapping parent calls with code that operate before and after the parent call.