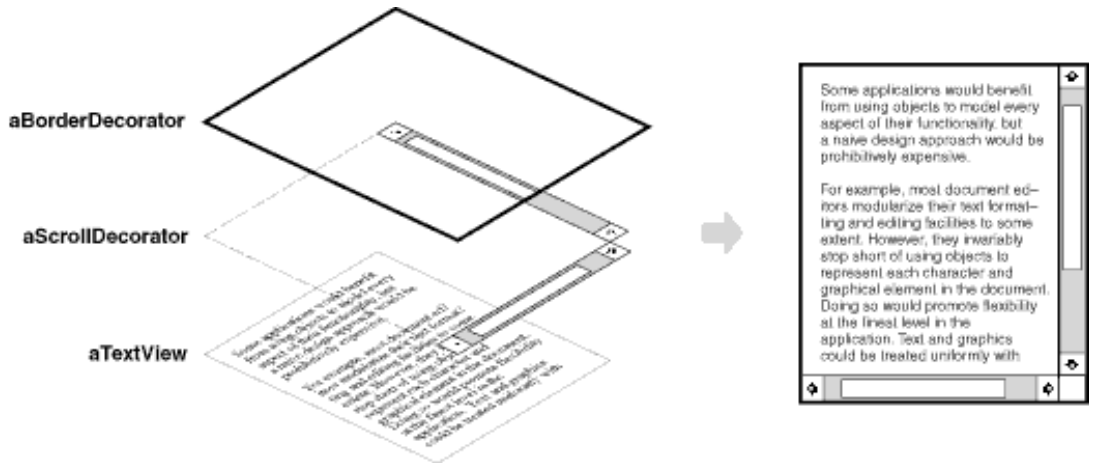
# CMPE 202

Gang of Four Design Patterns

## Decorator

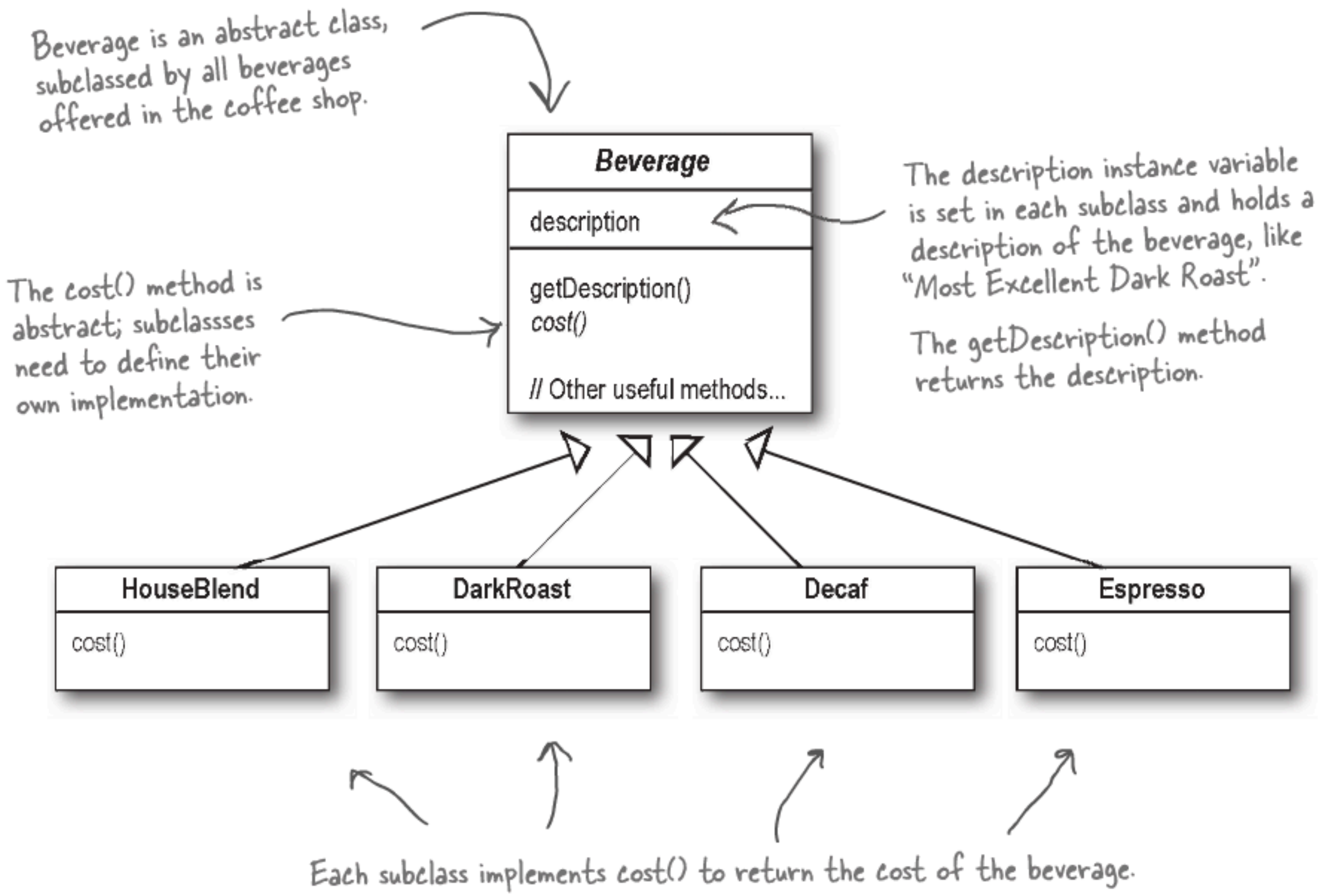
## Motivation

- Want to be able to add responsibilities to individual objects and not to all objects (i.e. the entire class)

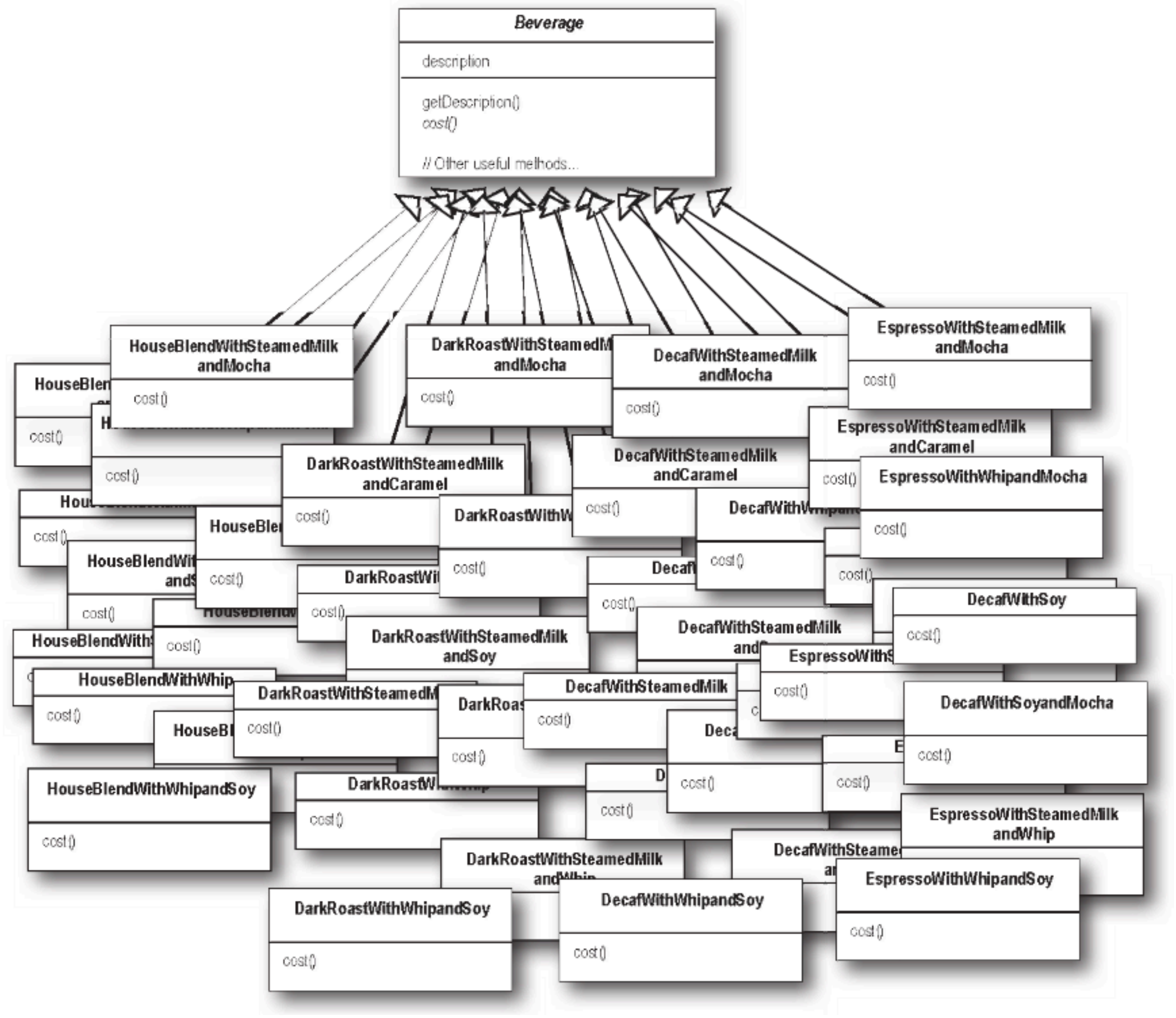


Some applications would benefit
from using objects to model every
aspect of their functionality, but
a naive design approach would be
prohibitively expensive.

For example, most document ed–
itors modularize their text format–
ting and editing facilities to some
extent. However, they invariably
stop short of using objects to
represent each character and
graphical element in the document.
Doing so would promote flexibility
at the finest level in the
application. Text and graphics
could be treated uniformly with


Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.
Beverage
description
getDescription()
cost()
// Other useful methods...
The description instance variable is set in each subclass and holds a description of the beverage, like "Most Excellent Dark Roast".
The getDescription() method returns the description.
The cost() method is abstract; subclassses need to define their own implementation.
HouseBlend
cost()
DarkRoast
cost()
Decaf
cost()
Espresso
cost()
Each subclass implements cost() to return the cost of the beverage.

**In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk. Starbuzz charges a bit for each of these, so they really need to get them built into their order system.**

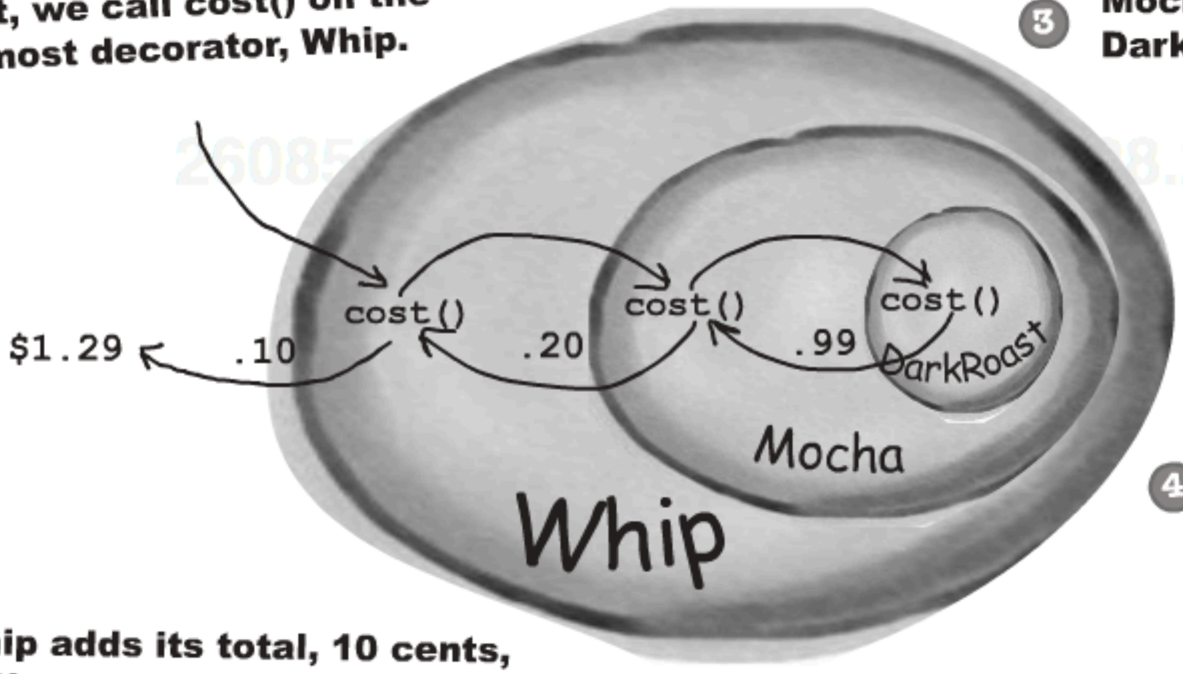


Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
cost()
DarkRoastWithSteamedMilk andMocha
cost()
DecafWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andCaramel
cost()
DarkRoastWithSteamedMilk andCaramel
cost()
DecafWithSteamedMilk andCaramel
cost()
EspressoWithWhipandMocha
cost()
DecafWithSoy
cost()
DarkRoastWithSteamedMilk andSoy
cost()
HouseBlendWithWhip
cost()
DecafWithSteamedMilk
cost()
DecafWithSoyandMocha
cost()
HouseBlendWithWhipandSoy
cost()
EspressoWithSteamedMilk andWhip
DarkRoastWithSteamedMilk andWhip
DarkRoastWithWhipandSoy
cost()
DecafWithWhipandSoy
cost()
EspressoWithWhipandSoy
cost()

2 Whip calls cost() on Mocha.

1 First, we call cost() on the outmost decorator, Whip.

3 Mocha calls cost() on DarkRoast.

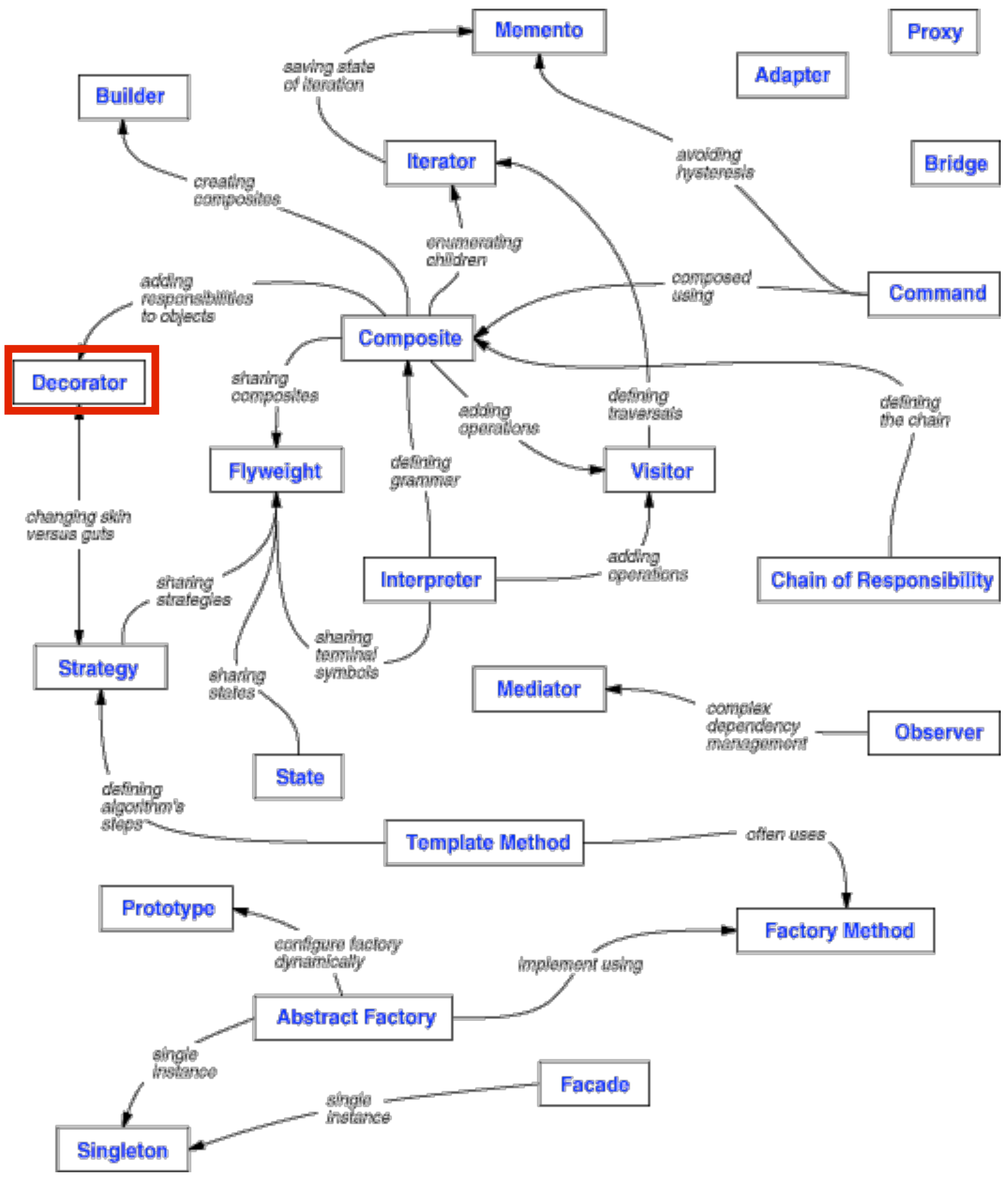


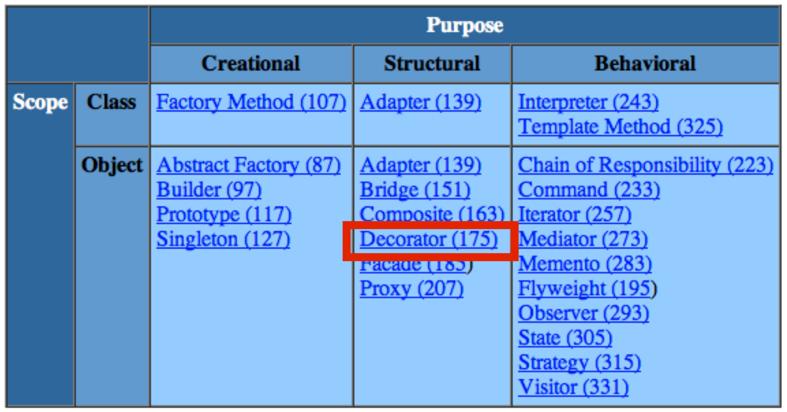
4 DarkRoast returns its cost, 99 cents.

5 Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.

5 Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

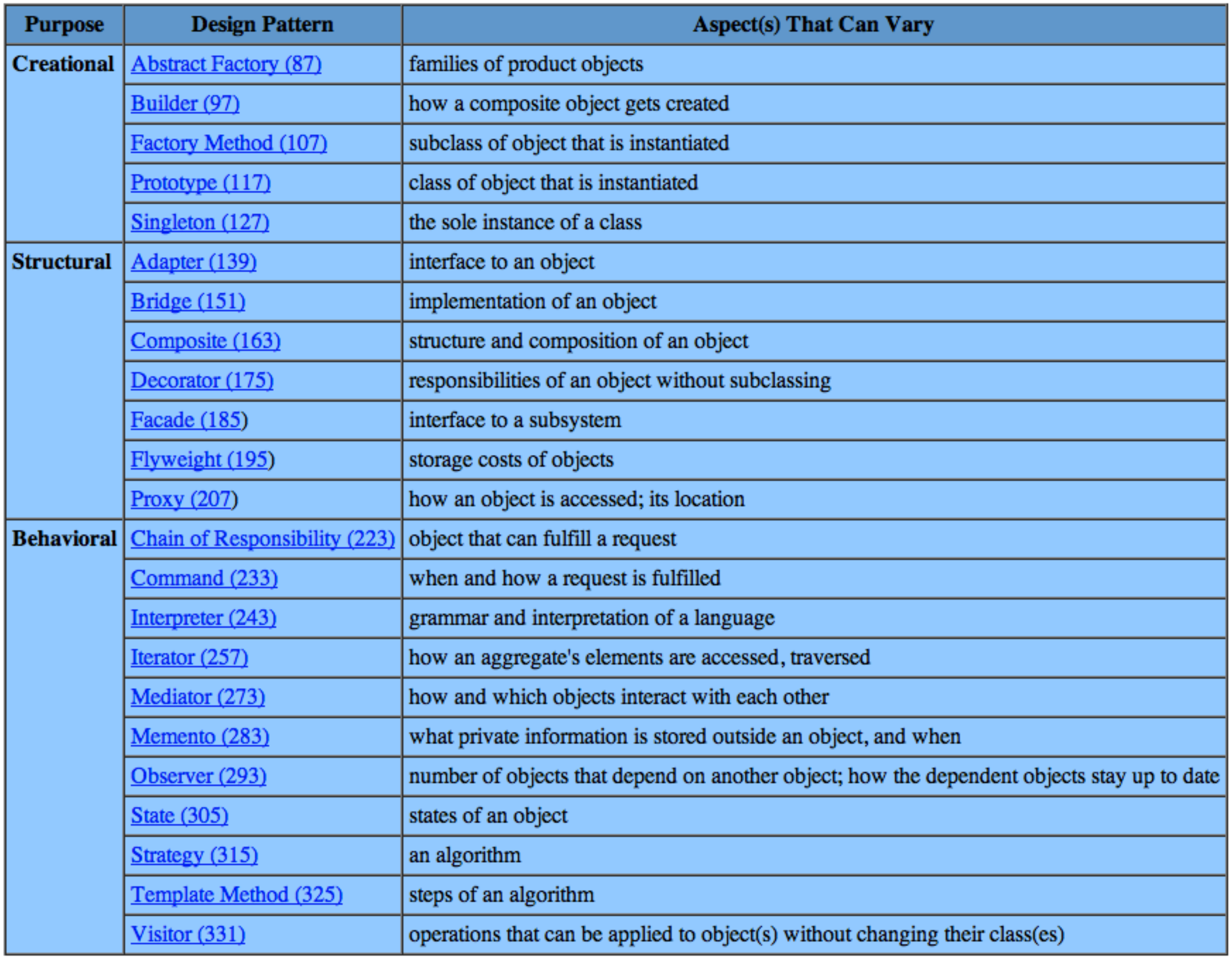
# Decorator





| | | Purpose | | |
|---|---|---|---|---|
| | | Creational | Structural | Behavioral |
| Scope | Class | Factory Method (107) | Adapter (139) | Interpreter (243)<br>Template Method (325) |
| | Object | Abstract Factory (87)<br>Builder (97)<br>Prototype (117)<br>Singleton (127) | Adapter (139)<br>Bridge (151)<br>Composite (163)<br>Decorator (175)<br>Facade (185)<br>Proxy (207) | Chain of Responsibility (223)<br>Command (233)<br>Iterator (257)<br>Mediator (273)<br>Memento (283)<br>Flyweight (195)<br>Observer (293)<br>State (305)<br>Strategy (315)<br>Visitor (331) |

# Design Pattern Catalog



| Purpose | Design Pattern | Aspect(s) That Can Vary |
|---|---|---|
| **Creational** | Abstract Factory (87) | families of product objects |
| | Builder (97) | how a composite object gets created |
| | Factory Method (107) | subclass of object that is instantiated |
| | Prototype (117) | class of object that is instantiated |
| | Singleton (127) | the sole instance of a class |
| **Structural** | Adapter (139) | interface to an object |
| | Bridge (151) | implementation of an object |
| | Composite (163) | structure and composition of an object |
| | Decorator (175) | responsibilities of an object without subclassing |
| | Facade (185) | interface to a subsystem |
| | Flyweight (195) | storage costs of objects |
| | Proxy (207) | how an object is accessed; its location |
| **Behavioral** | Chain of Responsibility (223) | object that can fulfill a request |
| | Command (233) | when and how a request is fulfilled |
| | Interpreter (243) | grammar and interpretation of a language |
| | Iterator (257) | how an aggregate's elements are accessed, traversed |
| | Mediator (273) | how and which objects interact with each other |
| | Memento (283) | what private information is stored outside an object, and when |
| | Observer (293) | number of objects that depend on another object; how the dependent objects stay up to date |
| | State (305) | states of an object |
| | Strategy (315) | an algorithm |
| | Template Method (325) | steps of an algorithm |
| | Visitor (331) | operations that can be applied to object(s) without changing their class(es) |

## Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

## Also Known As

Wrapper

## Applicability

Use Decorator

- to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- for responsibilities that can be withdrawn.
- when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

## Participants

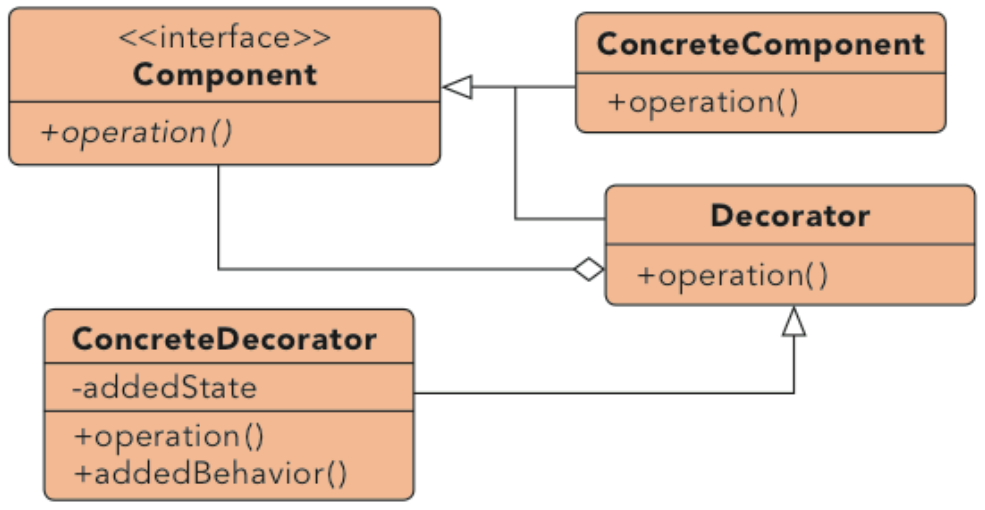
- **Component** (Interface)
  - defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent**
  - defines an object to which additional responsibilities can be attached.
- **Decorator**
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator**
  - 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.

## DECORATOR

Object Structural

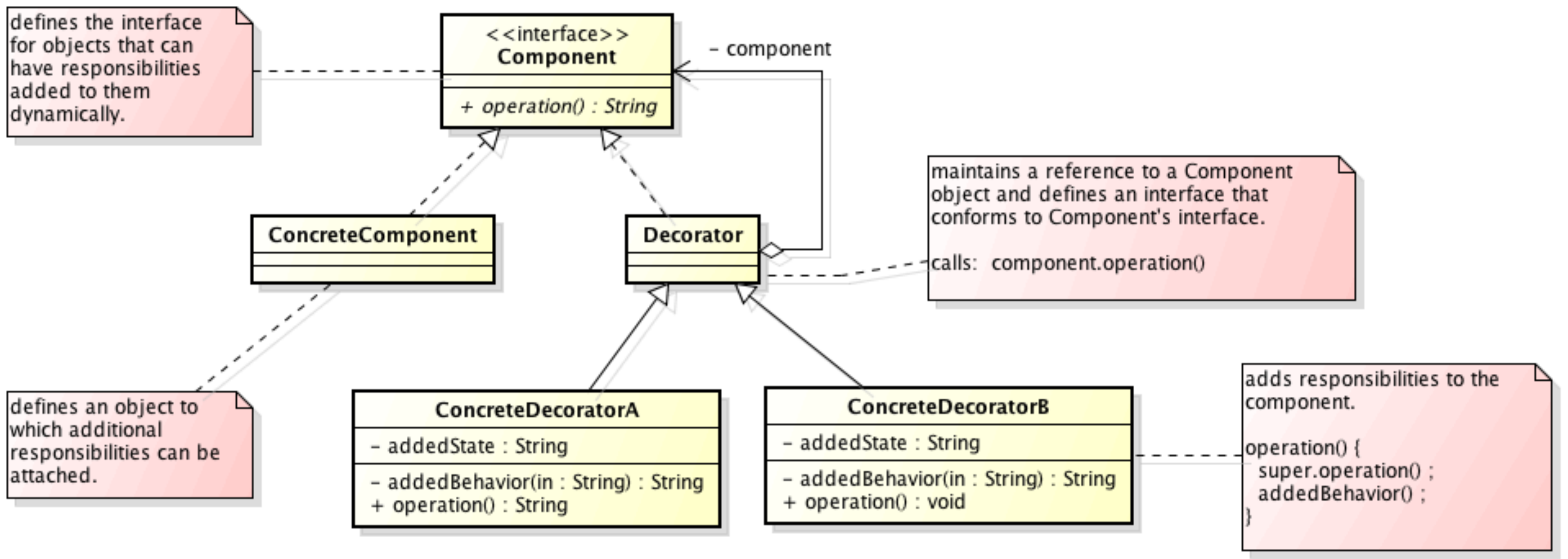


### Purpose

Allows for the dynamic wrapping of objects in order to modify their existing responsibilities and behaviors.

### Use When

- Object responsibilities and behaviors should be dynamically modifiable.
- Concrete implementations should be decoupled from responsibilities and behaviors.
- Subclassing to achieve modification is impractical or impossible.
- Specific functionality should not reside high in the object hierarchy.
- A lot of little objects surrounding a concrete implementation is acceptable.


defines the interface for objects that can have responsibilities added to them dynamically.
<<interface>>
Component
+ operation() : String
- component
ConcreteComponent
Decorator
maintains a reference to a Component object and defines an interface that conforms to Component's interface.
calls: component.operation()
defines an object to which additional responsibilities can be attached.
ConcreteDecoratorA
- addedState : String
- addedBehavior(in : String) : String
+ operation() : String
ConcreteDecoratorB
- addedState : String
- addedBehavior(in : String) : String
+ operation() : void
adds responsibilities to the component.
operation() {
super.operation() ;
addedBehavior() ;
}

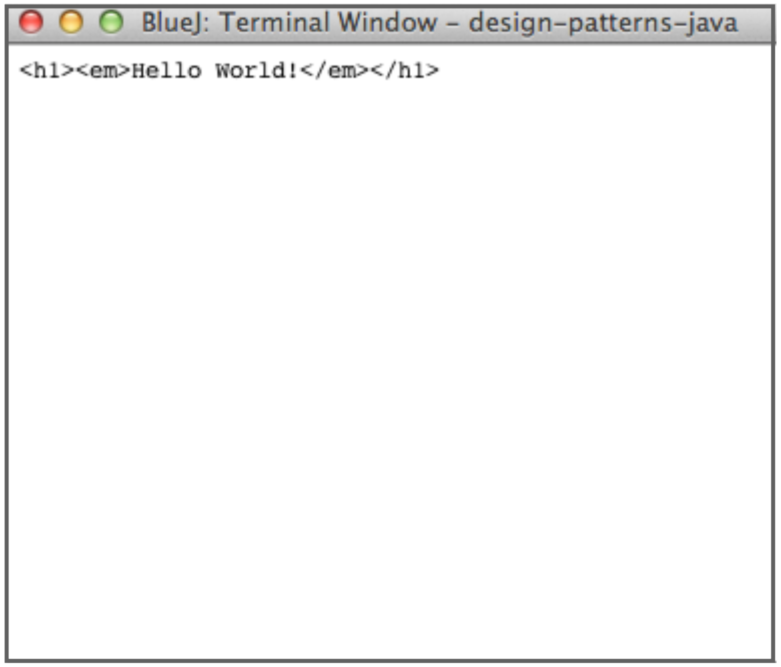
```java
public class Decorator implements Component {

    private Component component;

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

    public String operation()
    {
        return component.operation() ;
    }

}
```

```java
public class ConcreteComponent implements Component {

    public String operation() {
        return "Hello World!";
    }

}
```

```java
public class ConcreteDecoratorA extends Decorator {

    private String addedState;

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

    public String operation()
    {
        addedState = super.operation() ;
        return addedBehavior( addedState ) ;
    }

    private String addedBehavior(String in) {
        return "<em>" + addedState + "</em>" ;
    }

}
```

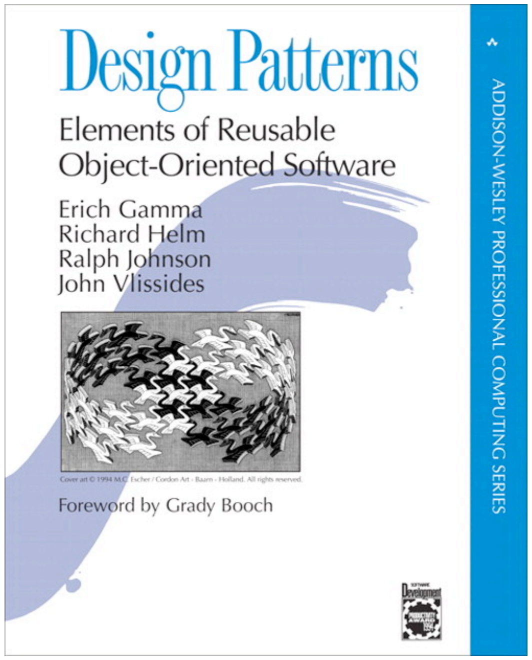
```java
public class ConcreteDecoratorB extends Decorator {

    private String addedState;

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

    public String operation()
    {
        addedState = super.operation() ;
        return addedBehavior( addedState ) ;
    }

    private String addedBehavior(String in) {
        return "<h1>" + addedState + "</h1>" ;
    }

}
```

```java
public class Tester {

    public static void runTest()
    {
        Component obj = new ConcreteDecoratorB( new ConcreteDecoratorA( new ConcreteComponent() ) ) ;
        String result = obj.operation() ;
        System.out.println( result );
    }
}
```



```
<h1><em>Hello World!</em></h1>
```

# Resources for this Tutorial



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

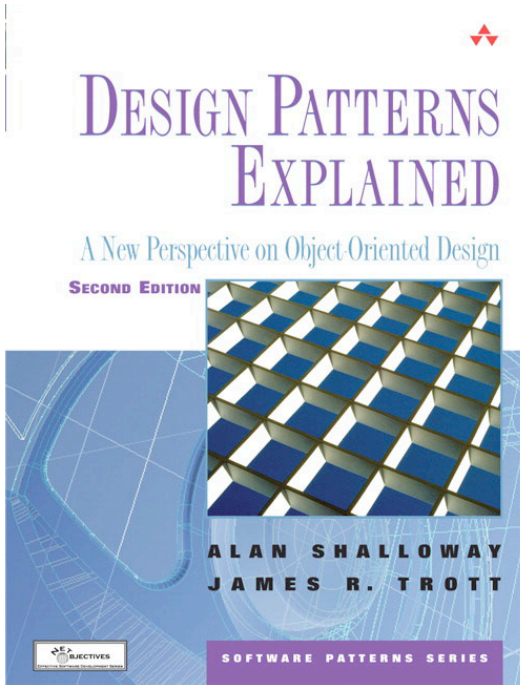
A Brain-Friendly Guide

Head First Design Patterns

Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance

O'REILLY

Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates



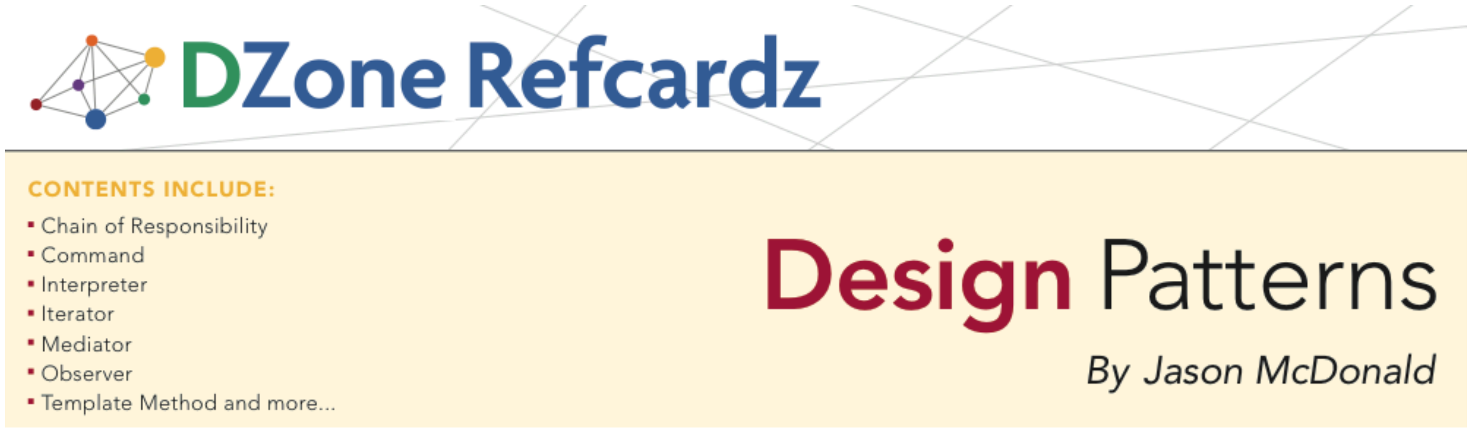
Design Patterns Explained

A New Perspective on Object-Oriented Design

Second Edition

Alan Shalloway
James R. Trott

Software Patterns Series



DZone Refcardz

CONTENTS INCLUDE:

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Observer
- Template Method and more...

Design Patterns

By Jason McDonald