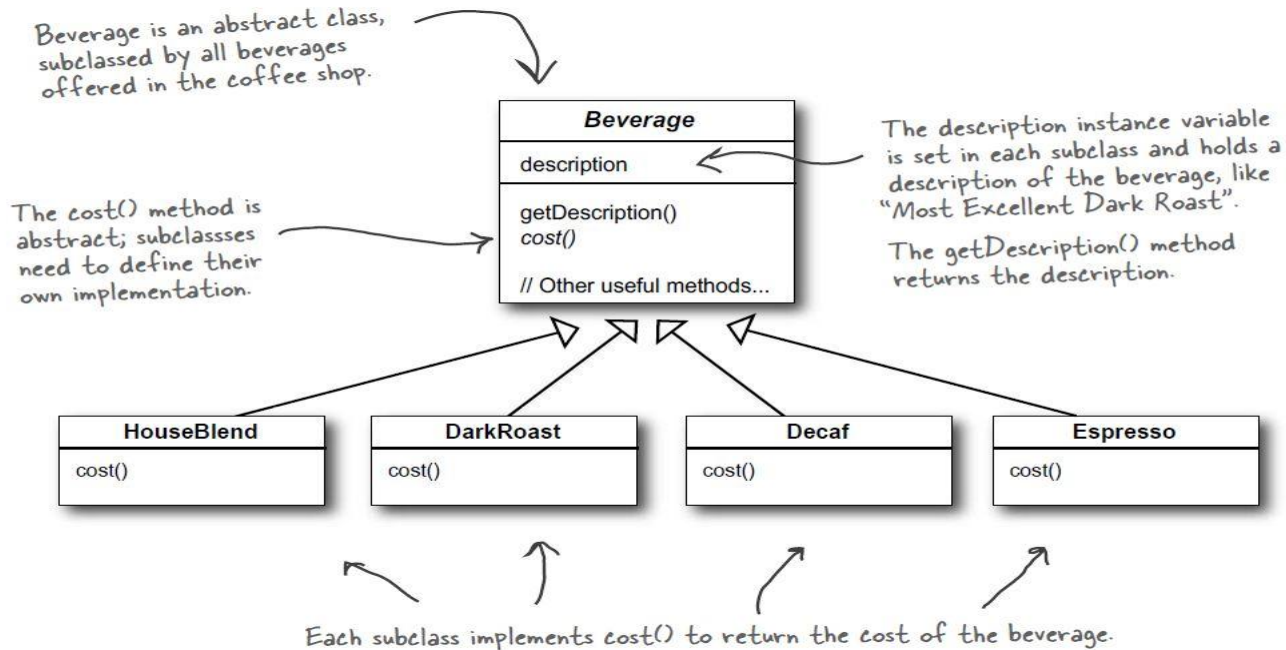


Decorator Pattern

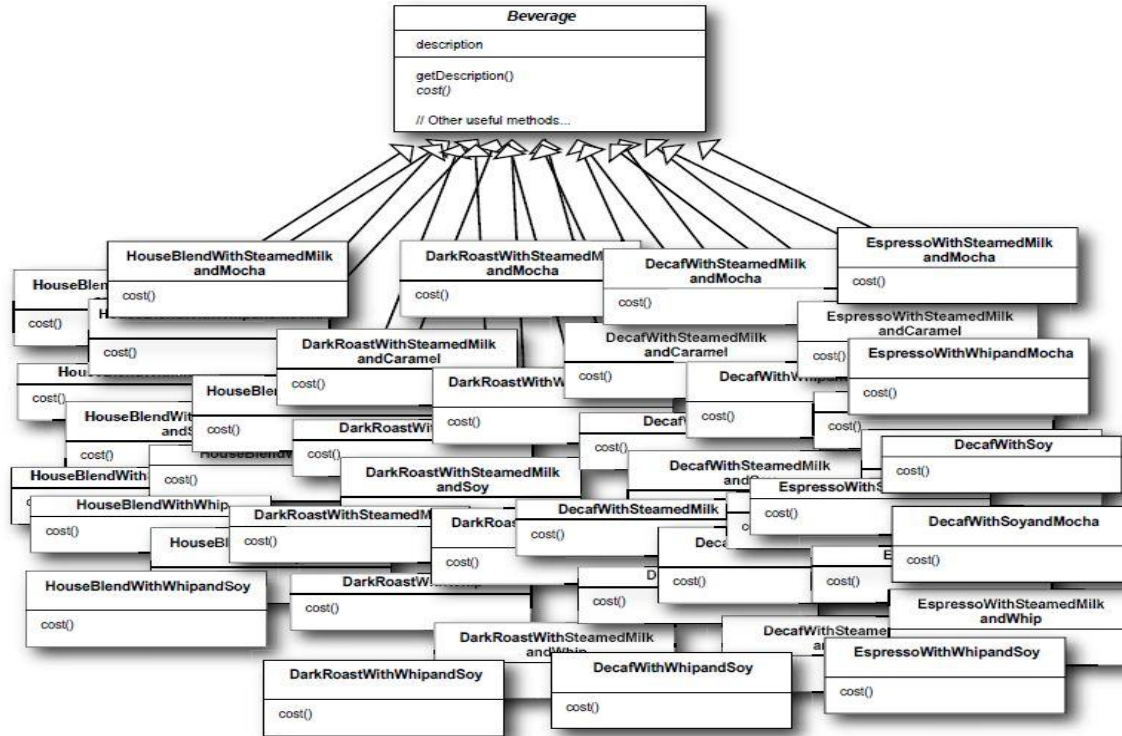
CSE776 – Design Patterns

Datan Xu, Vibhu Bharadwaj, Xu Zhang

Using Inheritance for Extending Functionality



Class Explosion Problem!!!

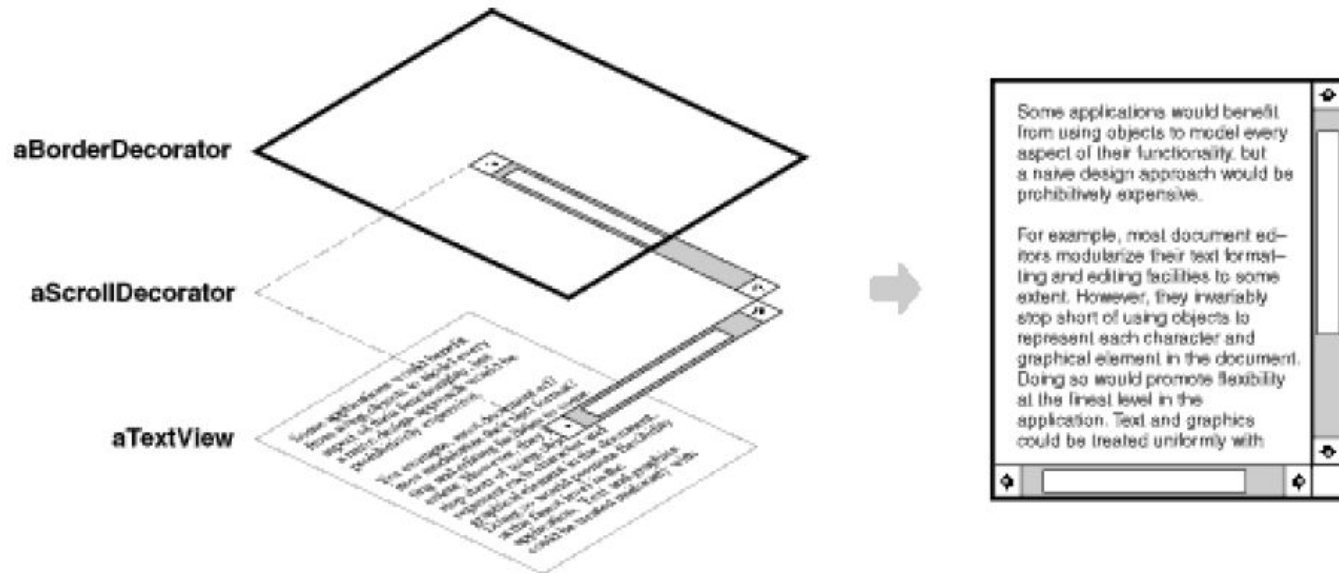


Intent

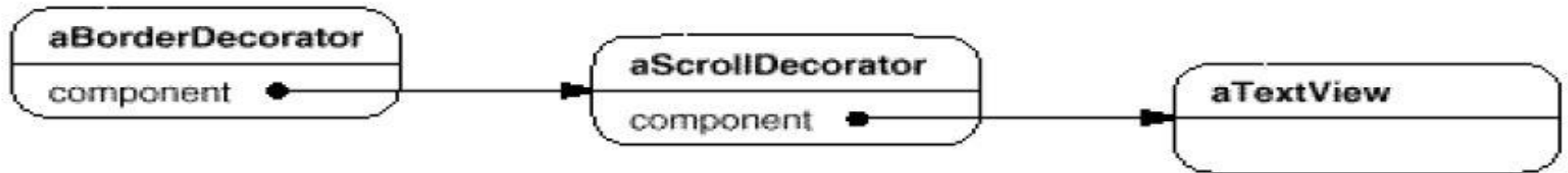
- ▶ “Attach additional responsibilities to an object **dynamically.**”
- ▶ Decorators provide a flexible alternative to subclassing for extending functionality.
- ▶ Also known as **Wrapper**



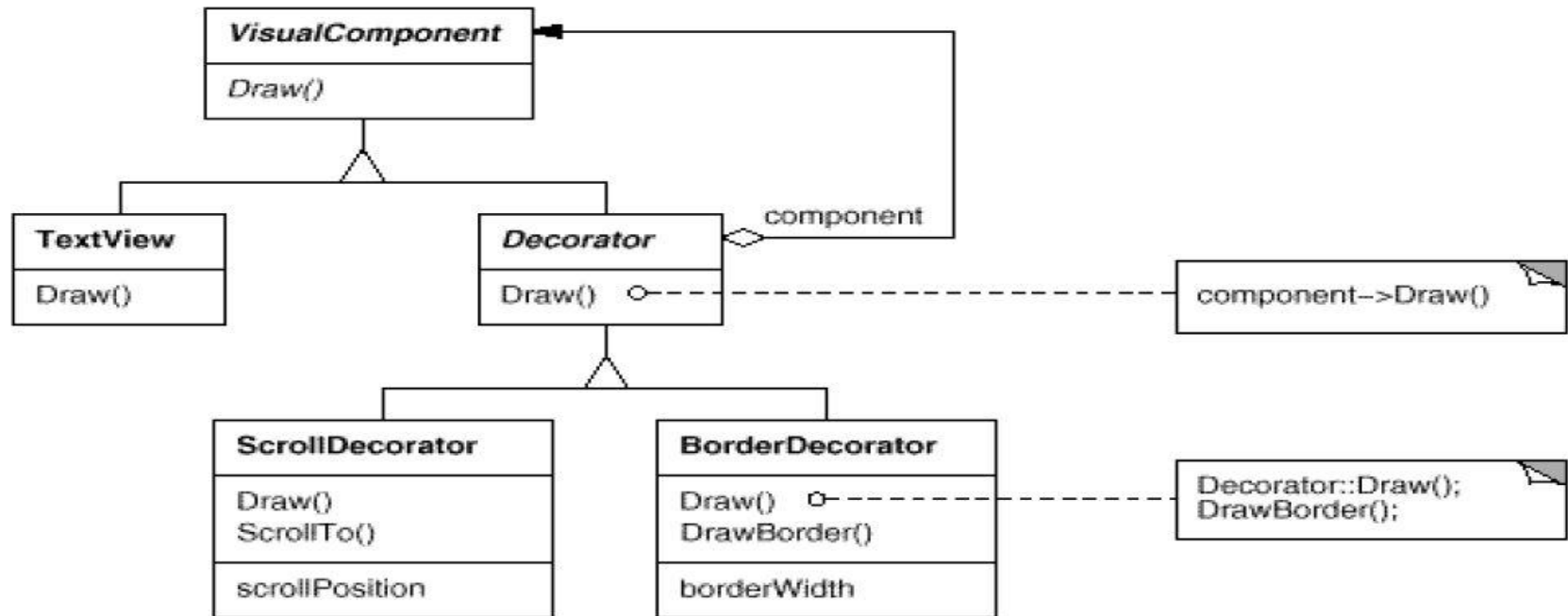
Motivation



Motivation



Motivation



Motivation

- ▶ Decorator Pattern addresses these problems:
 - ▶ VisualComponent is the abstract class for visual objects. It defines their drawing and event handling interface.
 - ▶ The important aspect of this pattern is that it lets decorators appear anywhere a VisualComponent can.
 - ▶ Clients generally can't tell the difference between a decorated component and an undecorated one, and so they don't depend at all on the decoration.



Forces

- ▶ The Open-Closed Principle--Design Principle
- ▶ Allow classes to be easily extended to incorporate new behavior without modifying existing code.
- ▶ Single Responsibility Principle - The decorator pattern is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern.



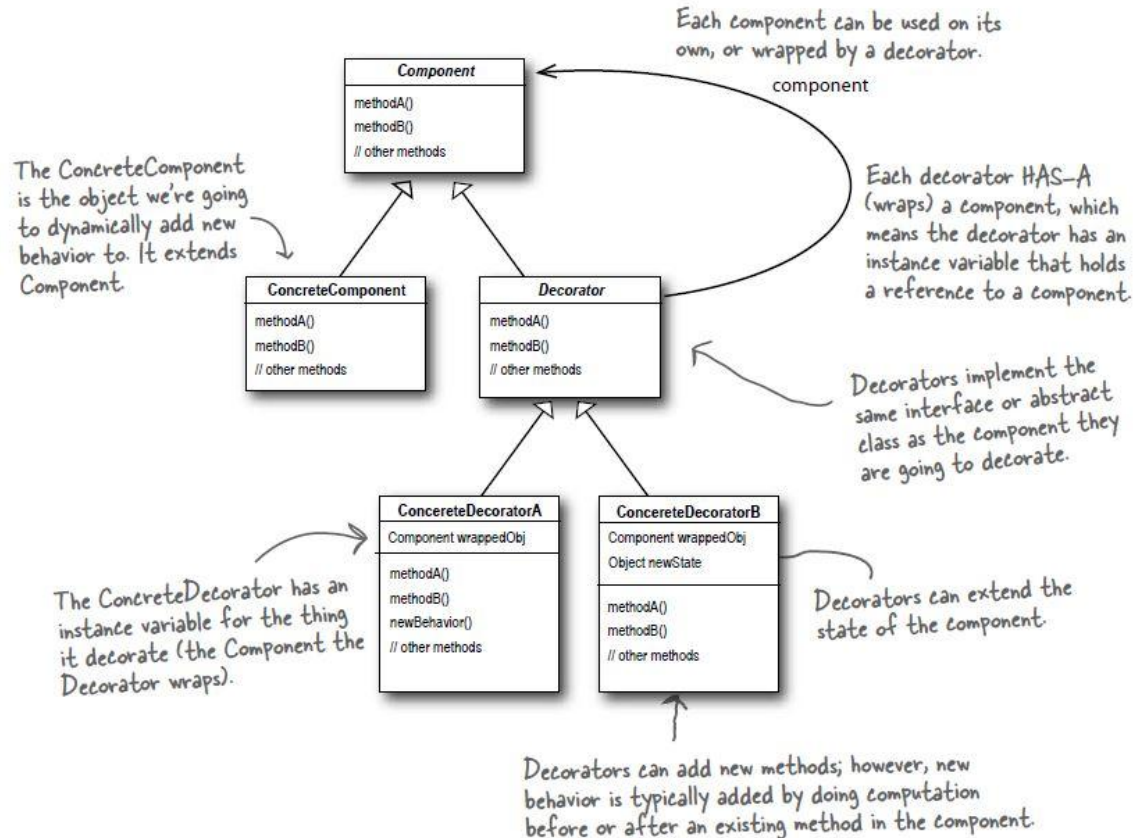
Applicability

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- ▶ for responsibilities that can be withdrawn.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.



Structure



Participants

- ▶ **Component (VisualComponent)**

- ▶ Defines the interface for objects that can have responsibilities added to them dynamically.

- ▶ **ConcreteComponent (TextView)**

- ▶ 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 (BorderDecorator, ScrollDecorator)**

- ▶ Adds responsibilities to the component.



Collaborations

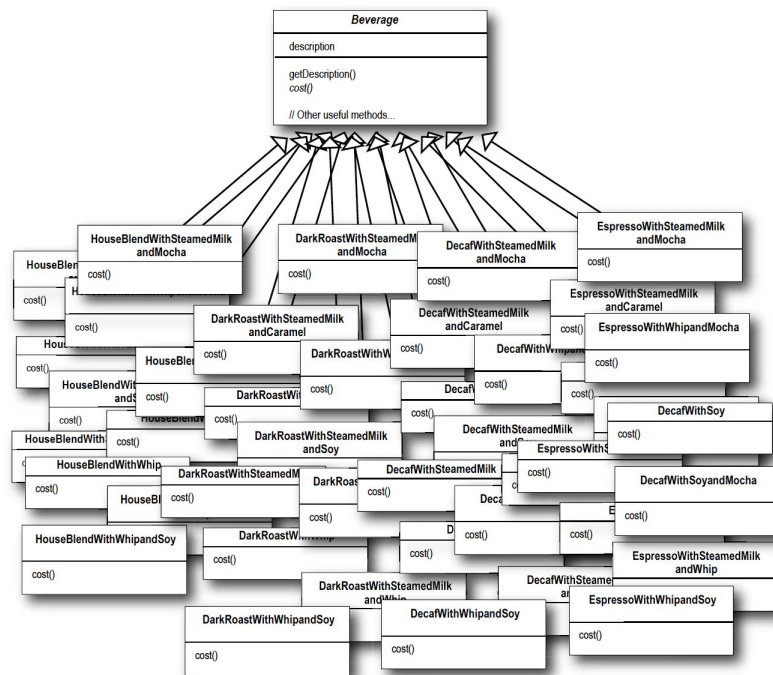
- ▶ “Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.”



Coding Implementation

Build Starbuzz Attempt 1

- Inheritance->explosion



Build Starbuzz Attempt 2

- Keep original coffee classes and add flags for every possible condiments
- Does it solve the explosion of classes problem?

```
4  class espresso {
5      public:
6          espresso() {
7              hasMilk_ = false;
8              hasChoco_ = false;
9          }
10
11         espresso(const bool & hasMilk, const bool & hasChoco)
12             :hasMilk_(hasMilk),hasChoco_(hasChoco){}
13
14         std::string getDescription() {
15             std::string descrip = "a shot of espresso";
16             if (hasMilk_) descrip += ", with milk";
17             if (hasChoco_) descrip += ", with chocolate syrup";
18             return descrip;
19         }
20
21         int cost() {
22             int cost = 3;
23             if (hasMilk_) cost += 2;
24             if (hasChoco_) cost += 1;
25             return cost;
26         }
27     private:
28         bool hasMilk_;
29         bool hasChoco_;
30 };
```

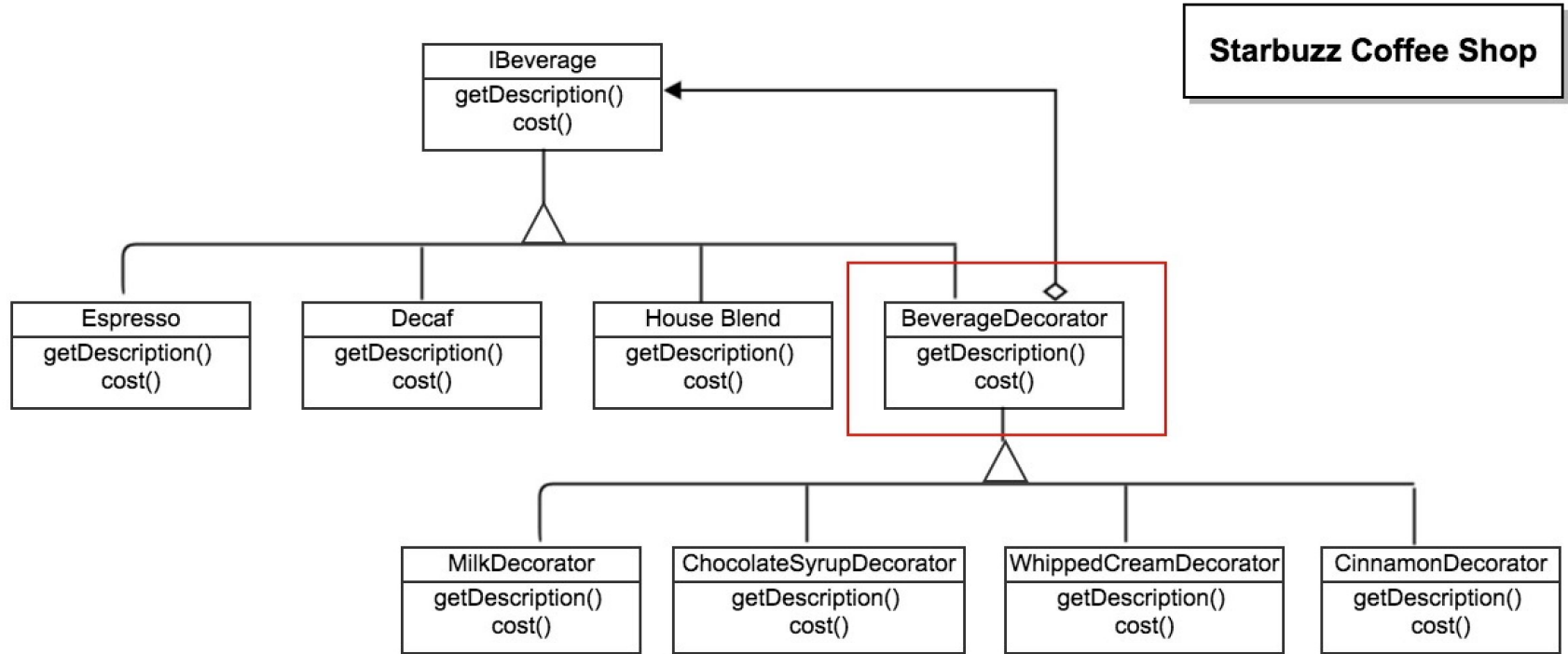
Build Starbuzz Attempt 2

- Keep original coffee classes and add flags for every possible condiments
- Does it solve the explosion of classes problem?

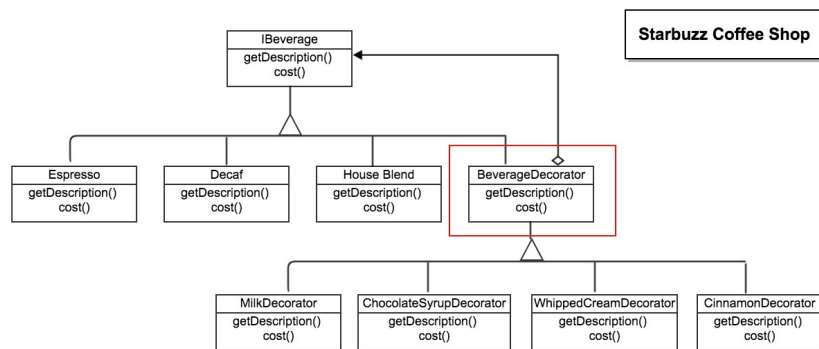
Violate OCP, multiple order of condiments not supported.

```
4  class espresso {
5      public:
6          espresso() {
7              hasMilk_ = false;
8              hasChoco_ = false;
9          }
10
11         espresso(const bool & hasMilk, const bool & hasChoco)
12             :hasMilk_(hasMilk),hasChoco_(hasChoco){}
13
14         std::string getDescription() {
15             std::string descrip = "a shot of espresso";
16             if (hasMilk_) descrip += ", with milk";
17             if (hasChoco_) descrip += ", with chocolate syrup";
18             return descrip;
19         }
20
21         int cost() {
22             int cost = 3;
23             if (hasMilk_) cost += 2;
24             if (hasChoco_) cost += 1;
25             return cost;
26         }
27     private:
28         bool hasMilk_;
29         bool hasChoco_;
30 };
```

Attempt 3: Enter Decorator



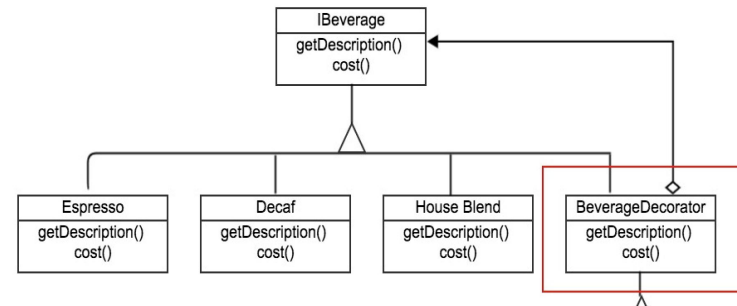
Key point



- Decorators have the same supertype as the objects being decorated.
- Decorator add behavior by delegating to the object it decorates, then add additional behavior.
- Can creating new decorator dynamically.

Starbuzz implementation

- IBeverage (component interface)



```
--
11 class IBeverage
12 {
13     public:
14         virtual std::string getDescription()=0;
15         virtual int cost() = 0;
16     protected:
17         std::string descrip_;
18 };
```

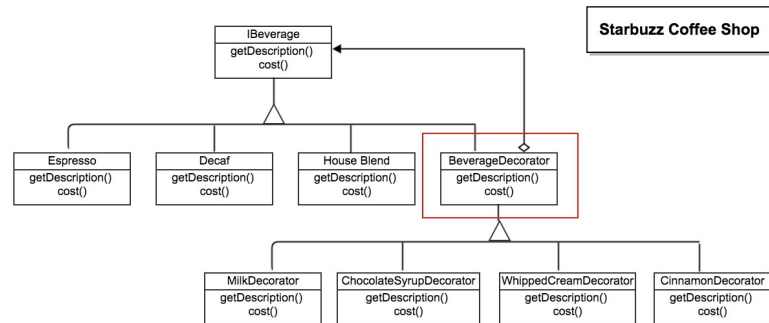
- Espresso (concrete component)

```
20 class Espresso : public IBeverage {
21     public:
22         Espresso() {
23             descrip_ = "Shot of espresso";
24         }
25         std::string getDescription() {
26             return descrip_;
27         }
28         int cost() {
29             return 3;
30         }
31 };
```

Starbuzz implementation

- BeverageDecorator (Decorator interface)

```
73 class BeverageDecorator: public IBeverage {
74     public:
75         BeverageDecorator(IBeverage * iBev) :iBev_(iBev) {}
76         std::string getDescription() {
77             return iBev_>getDescription();
78         }
79         int cost() {
80             return iBev_>cost();
81         }
82     protected:
83         IBeverage * iBev_;
84 };
```



- MilkDecorator (concrete decorator)

```
86 class MilkDecorator :public BeverageDecorator {
87     public:
88         MilkDecorator(IBeverage * iBev) :BeverageDecorator(iBev) {}
89         std::string getDescription() {
90             return iBev_>getDescription() + ", with milk";
91         }
92         int cost() {
93             return iBev_>cost() + 2;
94         }
95 };
```

Starbuzz implementation

- Client:

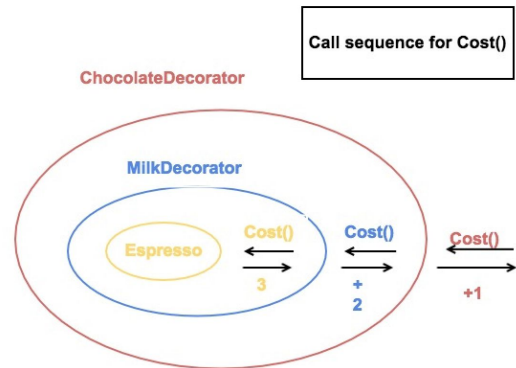
```
151 Espresso *espresso = new Espresso();
152 displayCoffee(espresso);
153
154 MilkDecorator* espWithMilk = new MilkDecorator(espresso);
155 displayCoffee(espWithMilk);
156
157 ChocolateSyrupDecorator * espWithChoco = new ChocolateSyrupDecorator(espWithMilk);
158 displayCoffee(espWithChoco);
159
```

- output

```
You ordered: Shot of espresso
cost of the drink is: $3

You ordered: Shot of espresso, with milk
cost of the drink is: $5

You ordered: Shot of espresso, with milk, with chocolate syrup
cost of the drink is: $6
```



The Pros and the Cons of a Decorator:

- Permutation - Provides dynamic pluggable solutions to deal with changing object responsibilities.
 - Simplicity - Better than multiple inheritance.
 - Extension - Easily add/remove/modify functionality without OCP violation.
 - Reuse - One decorator can be reused by all Concrete Component.
-
- ❖ Many little lookalikes (Single responsibility)
 - ❖ Instantiation - `wrap1(wrap2(wrap3(wrap4....(Component)....)))`
 - ❖ The cream is not the coffee! A decorator simply maintains a Component reference.
 - ❖ Overuse of decorators

Known Uses of a Decorator:

- Presentation - GUI design to decorate data and visualization preferences (different graphs, fonts, etc.)
- Dynamic usage - Select compression and decompression algorithms.
- Qualification/Adjective - Passing specifiers to HTTP headers (ex. Content length, referer, etc.)
- Extension - Extend a product line without altering the manufacturing process.

Related Patterns

- Adapter vs Decorator - The Interface!
- Composite vs Decorator - When in need of aggregation...
- Strategy vs Decorator - An inside-out solution vs an outside-in