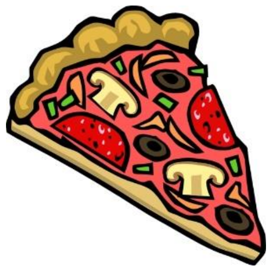
Decorator pattern

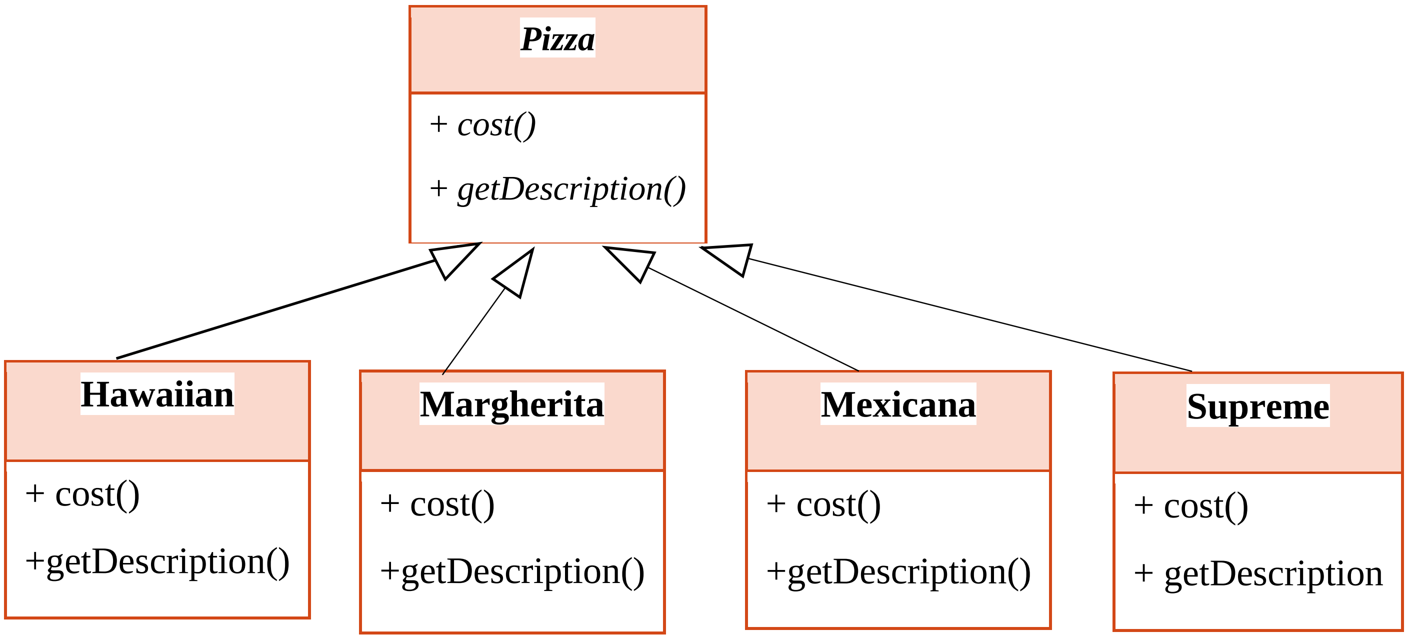
I. Problem

Welcome to domino's



Domino's has made a name for itself as the fastest growing pizza shop around. If you've seen one on your local corner, look across the street; you'll see another one. Because they've grown so quickly, they're scrambling to update their ordering systems to match their pizza offerings.

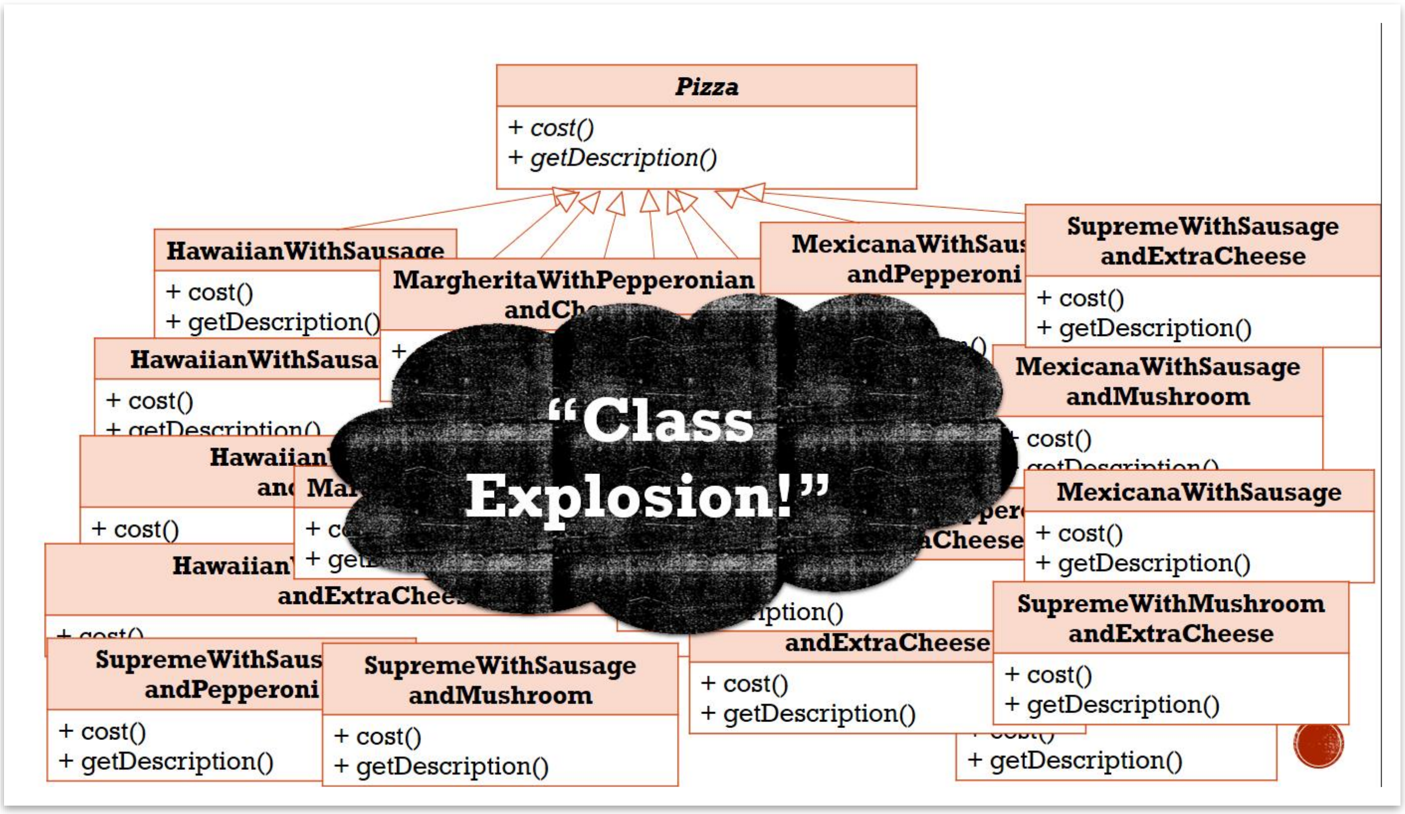
When they first went into business they designed their classes like this...



- Pizza is an abstract class, subclassed by all pizza offered in the pizza shop.
- The cost() method is abstract, subclasses need to define their own implementation.

In addition to your pizza, you can also ask for several toppings like sausage, pepperoni, mushrooms, extra cheese,… Domino's charges a bit for each of these, so they really need to get them built into their order system.

- *Here's their first attempt.*


Pizza
+ cost()
+ getDescription()
HawaiianWithSausage
MargheritaWithPepperonian andCh
MexicanaWithSaus andPepperoni
SupremeWithSausage andExtraCheese
HawaiianWithSausa
MexicanaWithSausage andMushroom
MexicanaWithSausage
SupremeWithMushroom andExtraCheese
SupremeWithSaus andPepperoni
SupremeWithSausage andMushroom
andExtraCheese
"Class Explosion!"

- Each cost method computes the cost of the coffee along with the other condiments in the order.

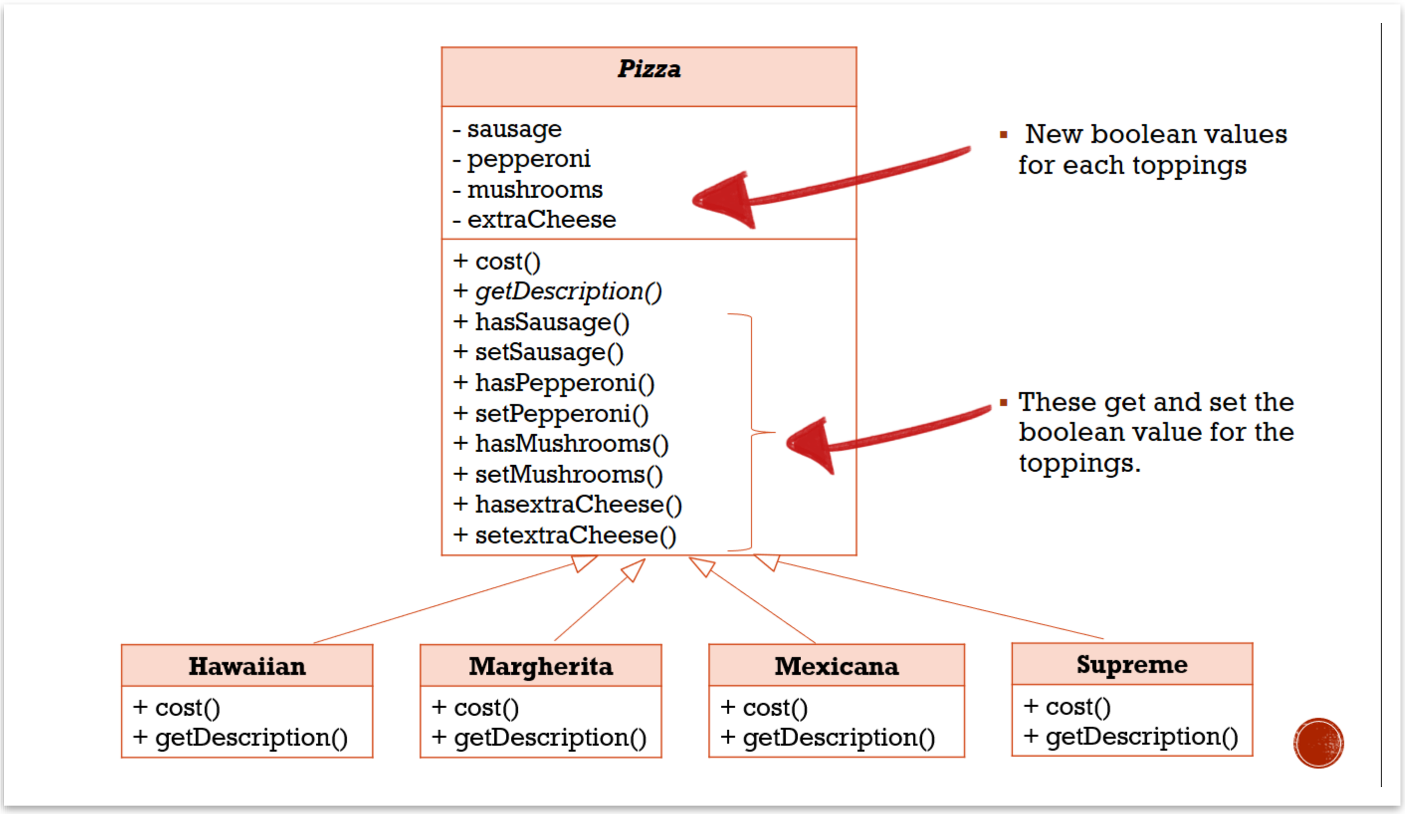
II. Alternate solution

- It's pretty obvious that Domino's has created a maintenance nightmare for themselves. What happens when the price of cheese goes up? What do they do when they add a new roast chicken topping?

- The application would go crazy maintaining all these concrete combinations of pizzas and toppings.

- Domino's team realize that all these subclasses solution is not going to work and it is a maintenance nightmare for themselves, so they start redesigning their system.

- They create instance variables to keep track of all toppings. So each subclass can set options which it requires.
- *Let's give it a try…*



- Now we'll implement cost() in pizza (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular Pizza instance. Subclasses will still override cost(), but they will also invoke the super version so that they can calculate the total cost of the basic pizza plus the costs of the added condiments.

III. Potential Problem

- There are some potential problems with this approach by thinking about how the design might need to change in the future.

- *But there are also some potential problems with this approach…*

1. Price changes for toppings will force us to alter existing code.

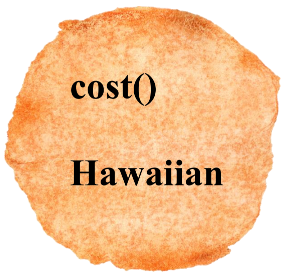
2. Whenever new topping options are introduced, they will force us to add new methods and alter the cost method in the superclass.
3. The super class has all the topping options, so for example ‘veggie mania’, can also have roast chicken topping, which may not be appropriate.
4. What If a customer wants a double extra cheese?

IV. Solve problem with decorator pattern.

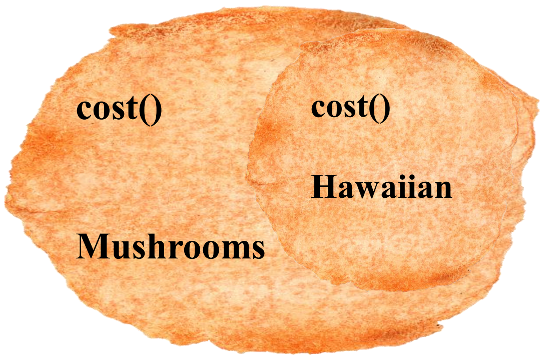
1. Constructing a pizza order with decorator

- We’ve seen that representing our pizza plus toppings pricing scheme with inheritance has not worked out very well. We get class explosions, rigid designs, or we add functionality to the base class that isn’t appropriate for some of the subclasses.
- Let’s implement Decorator Pattern. We will start with a pizza and “decorate” it with different condiments at run time. For example, if we’ve got an order for Hawaiian with mushrooms and pepperoni.
    1. **Take a Hawaiian object.**
    2. **Decorate it with mushrooms.**
    3. **Decorate it with pepperoni.**
    4. **Call the cost() method and relying on delegation to add cost of all toppings and pizza.**
- *Think of decorator objects as “wrappers.” Let’s see how this works...*

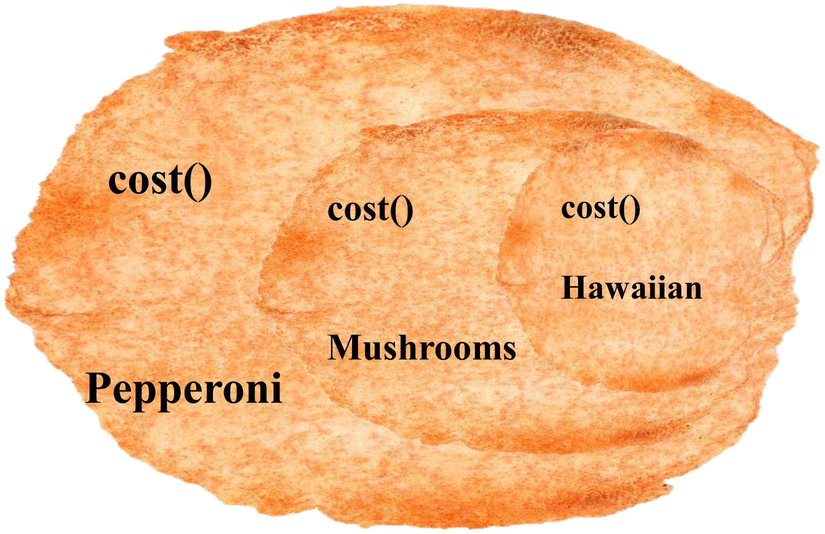
1. We start with our Hawaiian object.



2. The customer wants mushroom, so we create a mushroom object and wrap it around the Hawaiian.

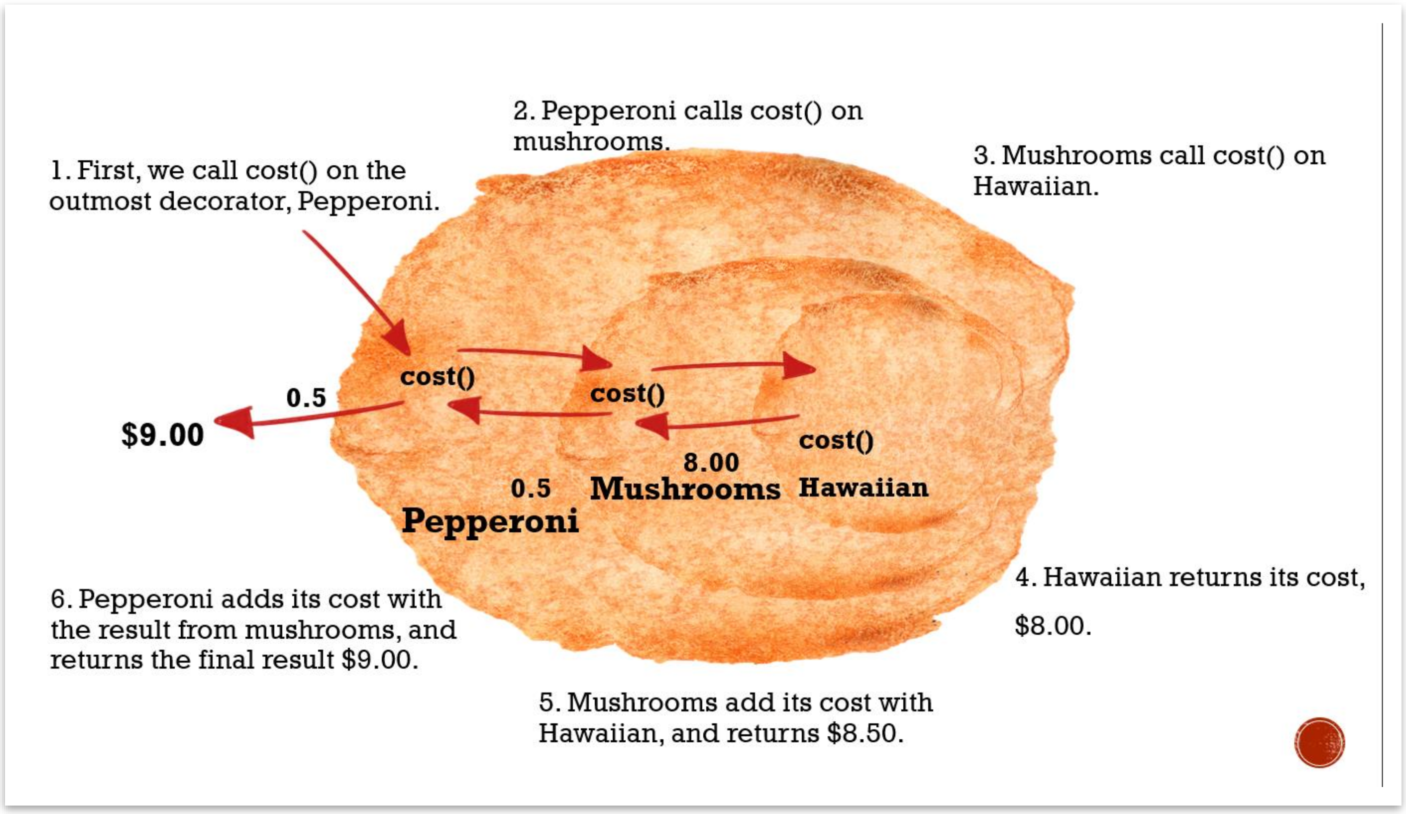


3. **The customer also wants Pepperoni, so we create a Pepperoni decorator and wrap Mushrooms with it.**



- A Hawaiian wrapped in Mushrooms and Pepperoni is still a pizza and we can do anything with it like we can do with a Hawaiian, including call its cost() method.

4. **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Pepperoni, and Pepperoni is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Pepperoni.**


1. First, we call cost() on the outmost decorator, Pepperoni.
2. Pepperoni calls cost() on mushrooms.
3. Mushrooms call cost() on Hawaiian.
4. Hawaiian returns its cost, $8.00.
5. Mushrooms add its cost with Hawaiian, and returns $8.50.
6. Pepperoni adds its cost with the result from mushrooms, and returns the final result $9.00.
cost()
cost()
cost()
$9.00
0.5
8.00
0.5
Pepperoni
Mushrooms
Hawaiian

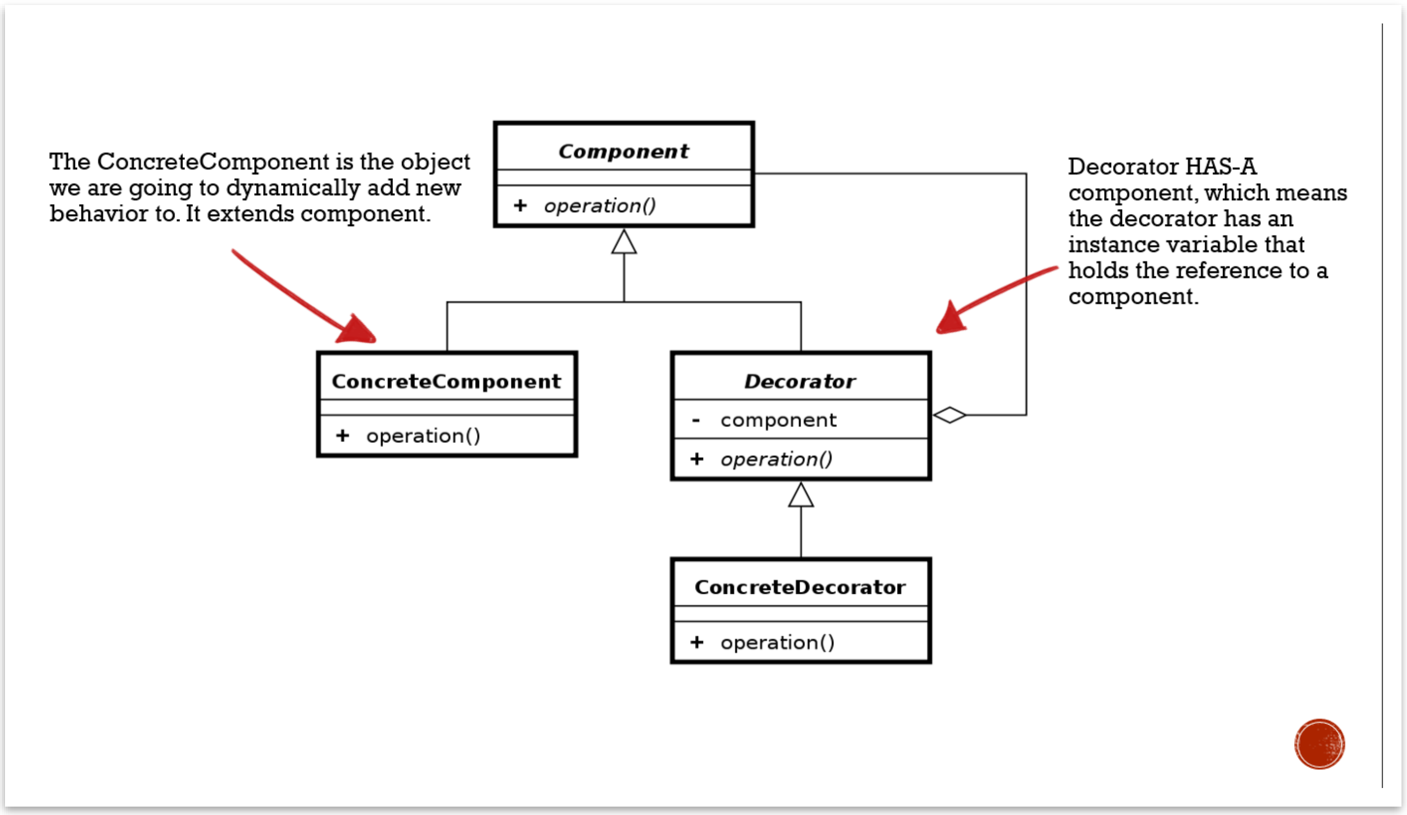
2. Some key points for decorator pattern

- Decorators have the same base type as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same super type as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.
- We can decorate objects dynamically at runtime with as many decorators as we like.

3. Decorator Pattern's Definition

- Decorator: Don't alter interface, but add responsibility.
- Decorator pattern allows a user to add new functionality to an existing object without altering its structure

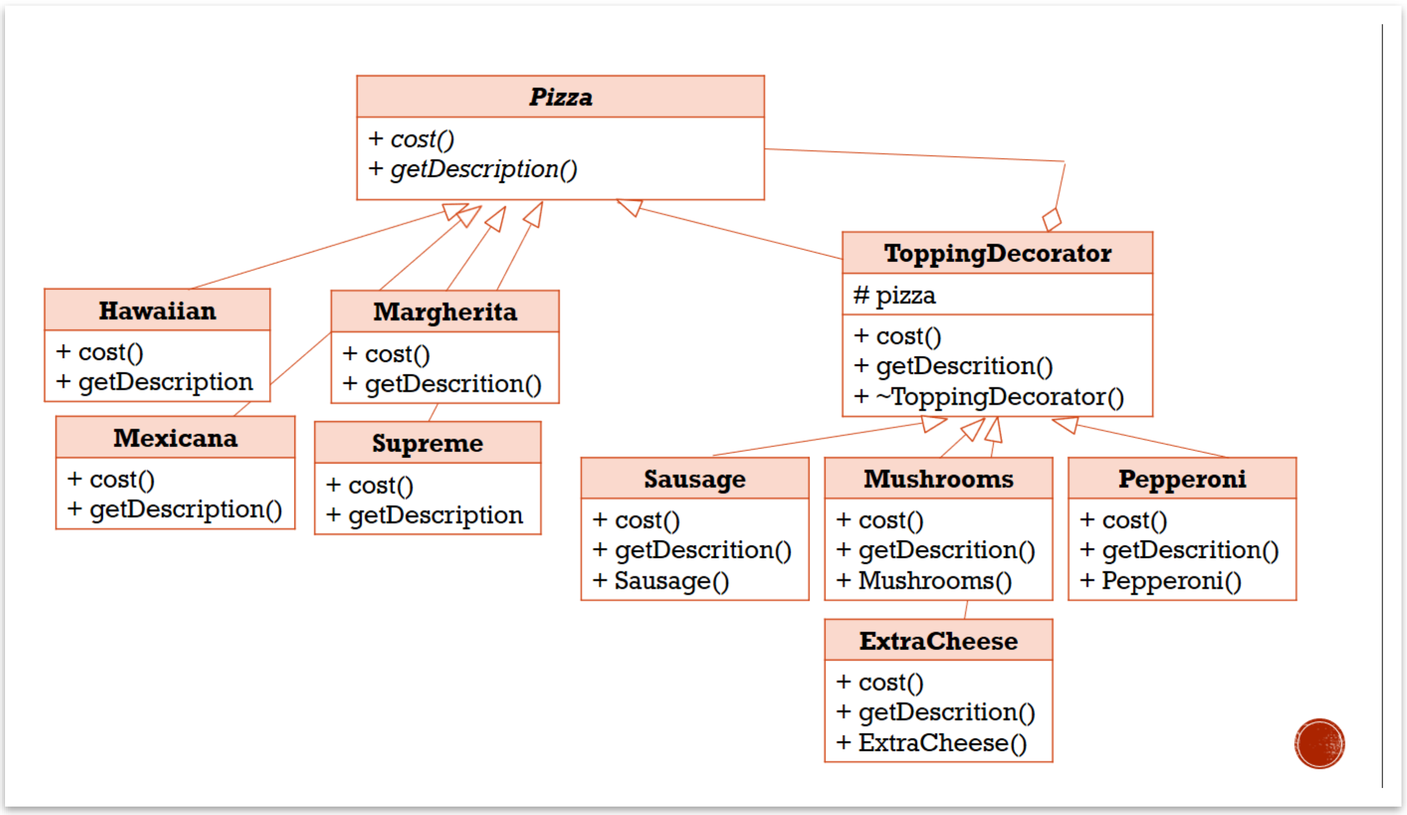
- This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.
- *Let's take a look at the class diagram…*


The ConcreteComponent is the object we are going to dynamically add new behavior to. It extends component.
Component
+ operation()
Decorator HAS-A component, which means the decorator has an instance variable that holds the reference to a component.
ConcreteComponent
+ operation()
Decorator
- component
+ operation()
ConcreteDecorator
+ operation()

https://en.wikipedia.org/wiki/Decorator_pattern

4. Decorating our Pizza

Let's work our Domino's pizza into this class diagram…



V. Writing the Domino's code

1. Coding abstract class

Let's start with the Pizza class, which is an abstract class:

```
class Pizza
{
     public:
     virtual string getDescription()=0;
     virtual float cost() = 0;
     private:
};
```

Let's implement the abstract class for the Topping Decorator:

```
class ToppingDecorator: public Pizza
{
public:
     ToppingDecorator(Pizza* pizza):pizza(pizza){};
     virtual string getDescription()=0;
     virtual float cost()=0;
     ToppingDecorator();
     ~ToppingDecorator();
protected:
     Pizza* pizza;
};
```

2. Coding Pizza

Now we've got our base classes, let's implement some pizzas.

```
class Hawaiian: public Pizza
{
public:
     float cost()
     {
        return 8.0;
     }
     string getDescription()
     {
        return "Hawaiian Pizza";
     }
     Hawaiian();
     ~Hawaiian();
};
```

```cpp
class Margherita: public Pizza
{
public:
     float cost()
     {
        return 6.00;
     }
     string getDescription()
     {
        return “Margherita Pizza”;
     }
     Margherita();
     ~Margherita();
};

class Mexicana: public Pizza
{
public:
     float cost()
     {
        return 10.0;
     }
     string getDescription()
     {
        return “Mexicana Pizza”;
     }
     Mexicana();
     ~Mexicana();
};
```

```
class Supreme: public Pizza
{
public:
     float cost()
     {
        return 10.0;
     }
     string getDescription()
     {
        return “Supreme Pizza”;
     }
     Supreme();
     ~Supreme();
};
```

3. Coding condiments

It’s time to implement the concrete decorators. Here’s Sausage:

```
class Sausage: public ToppingDecorator
{
public:
   float cost()
   {
        return 0.5 + pizza->cost();
   }
   string getDescription()
   {
        return pizza->getDescription()+“, Sausage”;
   }
   Sausage(Pizza* pizza):ToppingDecorator(pizza)
   {}
   Sausage();
   ~Sausage();
};
```

Here ‘s Mushrooms:

```
class Mushrooms: public ToppingDecorator
{
public:
   string getDescription()
   {
        return pizza->getDescription()+ “, Mushrooms”;
   }
   float cost()
   {
        return 0.5 + pizza->cost();
   }
   Mushrooms(Pizza* pizza):ToppingDecorator(pizza)
   {}
   Mushrooms();
   ~Mushrooms();
};
```

Here’s Pepperoni:

```
class Pepperoni: public ToppingDecorator
{
public:
   float cost()
   {
        return 0.5 + pizza->cost();
   }
   string getDescription()
   {
        return pizza->getDescription()+ “, Pepperoni”;
   }
   Pepperoni(Pizza* pizza):ToppingDecorator(pizza)
   {}
   Pepperoni();
   ~Pepperoni();
};
```

4. Serving some pizzas

Here is some code to make orders:

```
void main()
{
	cout << "Domino's pizza." << endl;
	Pizza * pizza = new Margherita();
	cout << pizza->getDescription() << " $" << pizza->cost()
<< endl;
	Pizza* pizza1 = new Hawaiian();
	pizza1 = new Mushrooms(pizza1);
	pizza1 = new Pepperoni(pizza1);
	cout << pizza1->getDescription() <<" $"<<pizza1->cost()
<< endl;
}
```

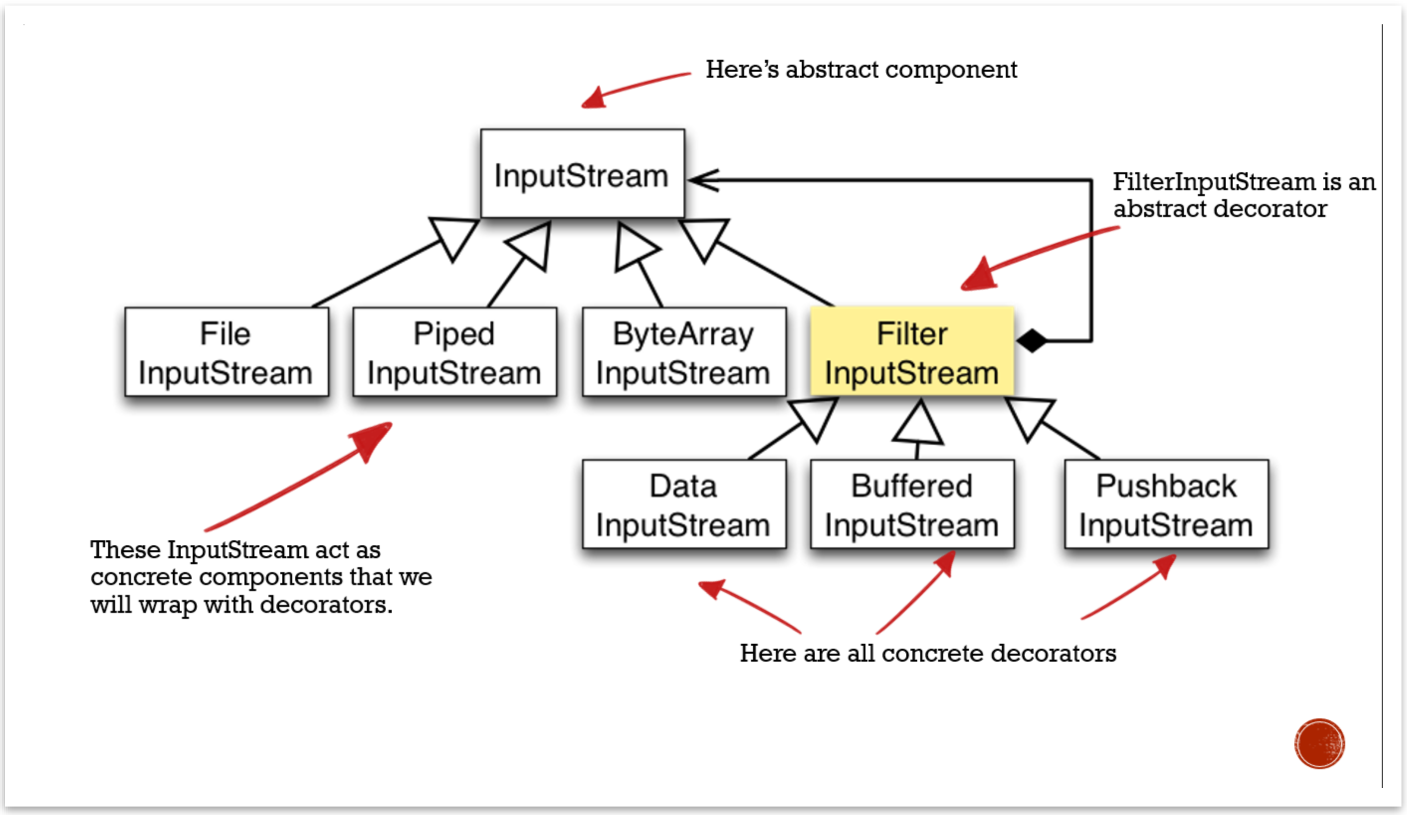
Here is the result:

```
Domino’s pizza.

Margherita Pizza $6

Hawaiian Pizza, Mushrooms, Pepperoni $9
```

VI. Real World Decorators

1. Java I/O

- The large number of classes in the java.io package is overwhelming. InputStream is an abstract class. Most concrete implementations like BufferedInputStream, GzipInputStream, ObjectInputStream, etc. have a constructor that takes an instance of the same abstract class. But now that you know the Decorator Pattern, the I/O classes should make more sense since the java.io package is largely based on Decorator.
- Here’s the class diagram of java.io…

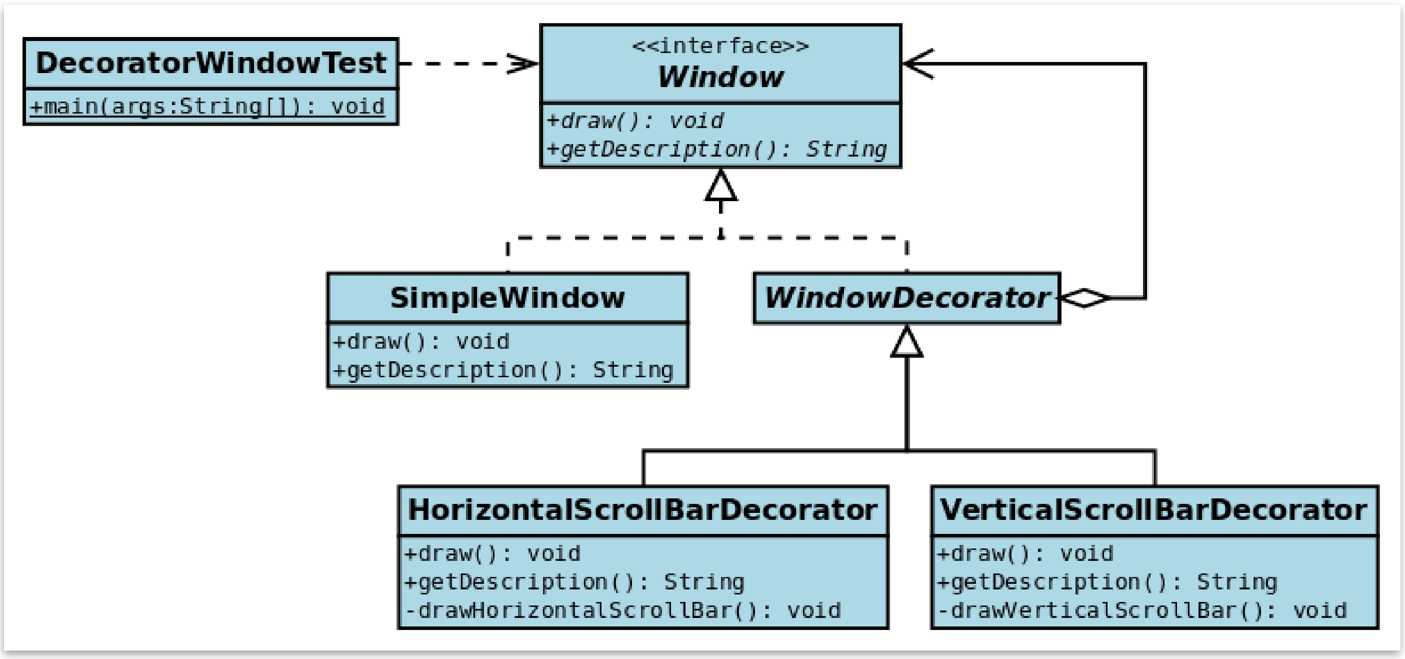

Here's abstract component
InputStream
FilterInputStream is an abstract decorator
File InputStream
Piped InputStream
ByteArray InputStream
Filter InputStream
Data InputStream
Buffered InputStream
Pushback InputStream
These InputStream act as concrete components that we will wrap with decorators.
Here are all concrete decorators

2. Window in a window system

- As an example, consider a window in a windowing system. To allow scrolling of the window's contents, one may wish to add horizontal or vertical scrollbars to it. Assume windows are represented by instances of the *Window* class, and assume this class has no functionality for adding scrollbars. One could create a *ScrollingWindowDecorator* that adds this functionality to existing *Window* objects.

Now, assume one also desires the ability to add borders to windows.

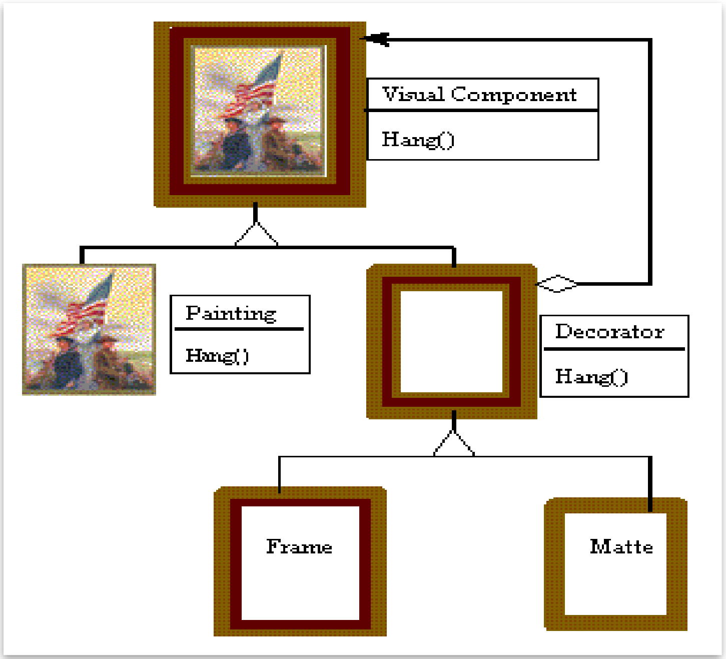
- We simply create a new *BorderedWindowDecorator*—at runtime, we can decorate existing windows with the *ScrollingWindowDecorator* or the *BorderedWindowDecorator* or both, as we see fit.
- Here's the class diagram…


DecoratorWindowTest
+main(args:String[]): void
<<interface>>
Window
+draw(): void
+getDescription(): String
SimpleWindow
+draw(): void
+getDescription(): String
WindowDecorator
HorizontalScrollBarDecorator
+draw(): void
+getDescription(): String
-drawHorizontalScrollBar(): void
VerticalScrollBarDecorator
+draw(): void
+getDescription(): String
-drawVerticalScrollBar(): void

https://en.wikipedia.org/wiki/Decorator_pattern

3. Picture's Frame

Picture often have frame added, and it is the frame which is actually hung on the wall. One or more mats are also optional. When complete, the painting, the mat(s), and the frame from a single visual component.



http://www.vincehuston.org/dp/real_demos.html

VII. Pros and Cons

1. Advantages

1. **More flexibility than static inheritance.**
   - The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance.
   - Responsibilities can be added and removed at run-time simply by attaching and detaching them. In contrast, inheritance requires creating a new class for each additional responsibility. This gives rise to many classes and increases the complexity of a system.
   - Decorators also make it easy to add a property twice.
2. **Avoids feature-laden classes with all the option in its.**
   - Decorator offers an approach to adding responsibilities. Instead of trying to support all features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.
   - Functionality can be composed from simple pieces. As a result, an application needn't pay for features it doesn't use.
   - It's also easy to define new kinds of Decorators independently from the classes of objects they extend. Extending a complex class tends to expose details unrelated to the responsibilities you're adding.

2. Disadvantages

1. **A decorator and its component aren't identical.**
   - A decorator acts as a transparent enclosure. But from an object identity point of view, a decorated component is not identical to the component itself. Hence you shouldn't rely on object identity when you use decorators.
2. **Lots of little objects.**
   - A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables.
   - Although these systems are easy to customize by those who understand them, they can be hard to learn and debug. Decorators can result in many small objects in our design, and overuse can be complex.

Reference

- Head first design pattern - Elisabeth Freeman Eric Freeman.
- Design Patterns - The "Gang of Four"
- http://stackoverflow.com/questions/2707401/please-help-me-understand-the-decorator-pattern-with-a-real-world-example
- http://conceptf1.blogspot.com/2016/01/decorator-design-pattern.html
- https://sourcemaking.com/design_patterns/decorator