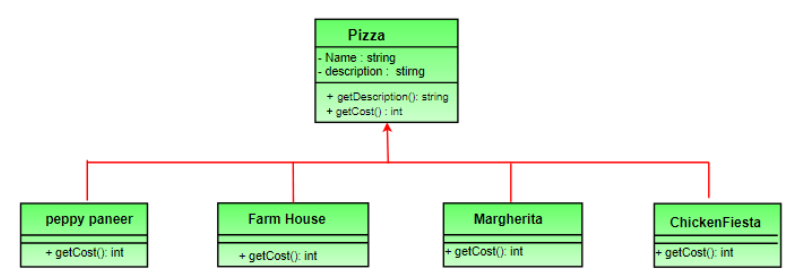
Decorator Pattern

Suppose we are building an application for a pizza store and we need to model their pizza classes. Assume they offer four types of pizzas namely Peppy Paneer, Farmhouse, Margherita and Chicken Fiesta. Initially we just use inheritance and abstract out the common functionality in a **Pizza**class

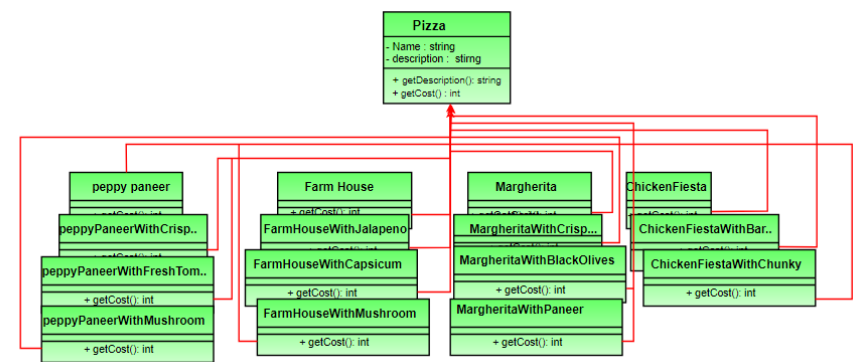


Each pizza has different cost. We have overridden the getCost() in the subclasses to find the appropriate cost. Now suppose a new requirement, in addition to a pizza, customer can also ask for several toppings such as Fresh Tomato, Paneer, Jalapeno, Capsicum, Barbeque, etc. Let us think about for some time that how we accommodate changes in the above classes so that customer can choose pizza with toppings and we get the total cost of pizza and toppings the customer chooses.

Let us look at various options.

**Option 1**

Create a new subclass for every topping with a pizza. The class diagram would look like:

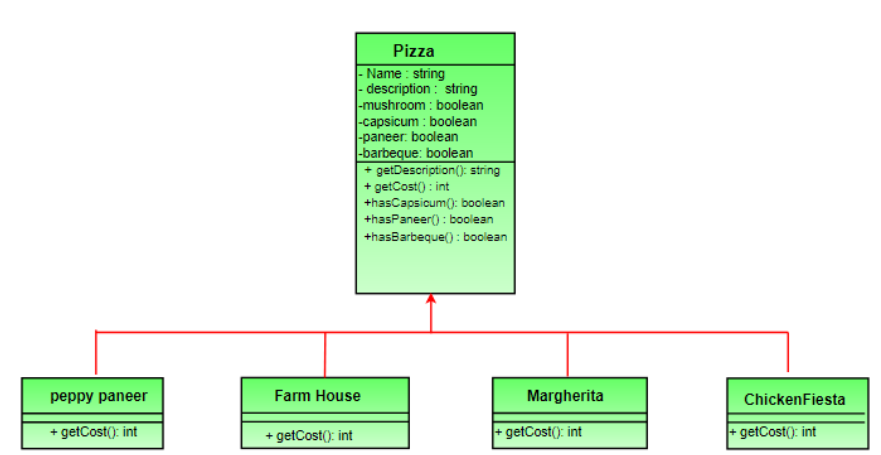


This looks very complex. There are way too many classes and is a maintenance nightmare. Also if we want to add a new topping or pizza we have to add so many classes. This is obviously very bad design.

Problem in this approach:

We want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

**Option 2:**  
Let’s add instance variables to pizza base class to represent whether or not each pizza has a topping. The class diagram would look like:



The getCost() of super class calculates the costs for all the toppings while the one in the subclass adds the cost of that specific pizza.

// Sample getCost() in super class

public int getCost()

{

int totalToppingsCost = 0;

if (hasJalapeno() )

totalToppingsCost += jalapenoCost;

if (hasCapsicum() )

totalToppingsCost += capsicumCost;

// similarly for other toppings

return totalToppingsCost;

}

// Sample getCost() in subclass

public int getCost()

{

// 100 for Margherita and super.getCost()

// for toppings.

return super.getCost() + 100;

}

This design looks good at first but lets take a look at the problems associated with it.

* Price changes in toppings will lead to alteration in the existing code.
* New toppings will force us to add new methods and alter getCost() method in super class.
* For some pizzas, some toppings may not be appropriate yet the subclass inherits them.
* What if customer wants double capsicum or double cheese burst?

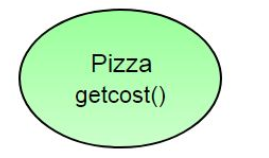
In short our design violates one of the most popular design principles – [***The Open-Closed Principle***](https://en.wikipedia.org/wiki/Open/closed_principle) which states that classes should be open for extension and closed for modification.

In the next set, we will be introducing Decorator Pattern and apply it to above problem.

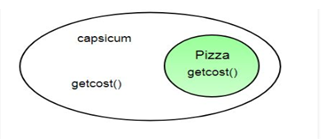
So, Solution is decorator pattern:

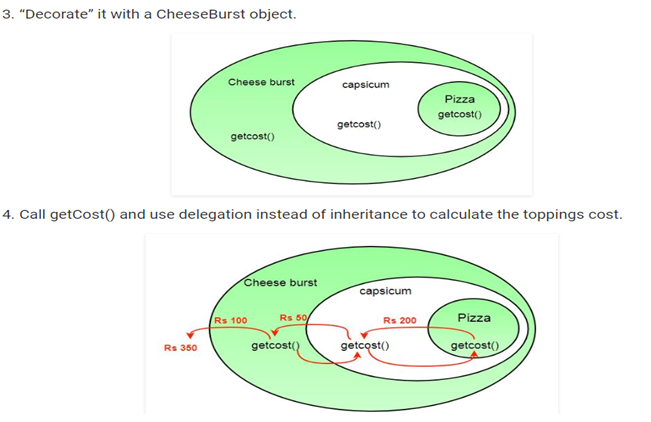
What we do now is take a pizza and “decorate” it with toppings at runtime:

1. Take a pizza object.



1. “Decorate” it with a Capsicum object.





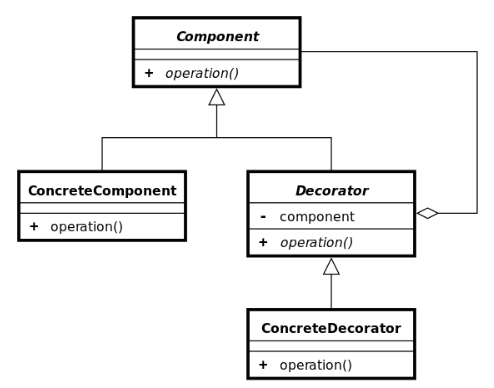
What we get in the end is a pizza with cheese burst and capsicum toppings. Visualize the “decorator” objects like wrappers. Here are some of the properties of decorators:

* Decorators have the same super type as the object they decorate.
* You can use multiple decorators to wrap an object.
* Since decorators have same type as object, we can pass around decorated object instead of original.
* We can decorate objects at runtime.

**Definition:**

The decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

**Class Diagram:**



* **Component** - Interface for objects that can have responsibilities added to them dynamically.
* **Concrete Component** - Defines an object to which additional responsibilities can be added.
* **Decorator** - Maintains a reference to a Component object and defines an interface that conforms to Component's interface.
* **Concrete Decorators** - Concrete Decorators extend the functionality of the component by adding state or adding behavior.
* Each component can be used on its own or may be wrapped by a decorator.
* Each decorator has an instance variable that holds the reference to component it decorates (HAS-A relationship).
* The Concrete Component is the object we are going to dynamically decorate.

**Advantages:**

* The decorator pattern can be used to make it possible to extend (decorate) the functionality of a certain object at runtime**.**
* The decorator pattern is an alternative to sub classing. Sub classing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at runtime for individual objects.
* Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.

**Disadvantages:**

* Decorators can complicate the process of instantiating the component because you not only have to instantiate the component, but wrap it in a number of decorators.
* It can be complicated to have decorators keep track of other decorators, because to look back into multiple layers of the decorator chain starts to push the decorator pattern beyond its true intent.

**Related Patterns**

* **Adapter Pattern** - A decorator is different from an adapter in that a decorator changes object's responsibilities, while an adapter changes an object interface.
* **Composite Pattern** - A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities.

**Steps:**

1. Create a "lowest common denominator" that makes classes interchangeable
2. Create a second level base class for optional functionality
3. "Core" class and "Decorator" class declare an "is a" relationship
4. Decorator class "has a" instance of the "lowest common denominator"
5. Decorator class delegates to the "has a" object
6. Create a Decorator derived class for each optional embellishment
7. Decorator derived classes delegate to base class AND add extra stuff
8. Client has the responsibility to compose desired configurations

Example 1:

*// 1. "lowest common denominator"*

**class** **Widget**

{

**public**:

**virtual** **void** draw() = 0;

};

**class** **TextField**: **public** Widget

{

*// 3. "Core" class & "is a"*

**int** width, height;

**public**:

TextField(**int** w, **int** h)

{

width = w;

height = h;

}

**void** draw()

{

cout << "TextField: " << width << ", " << height << '\n';

}

};

*// 2. 2nd level base class*

**class** **Decorator**: **public** Widget *// 4. "is a" relationship*

{

Widget \*wid; *// 4. "has a" relationship*

**public**:

Decorator(Widget \*w)

{

wid = w;

}

*/\*virtual\*/*

**void** draw()

{

wid->draw(); *// 5. Delegation*

}

};

**class** **BorderDecorator**: **public** Decorator

{

**public**:

*// 6. Optional embellishment*

BorderDecorator(Widget \*w): Decorator(w){}

*/\*virtual\*/*

**void** draw()

{

*// 7. Delegate to base class and add extra stuff*

Decorator::draw();

cout << " BorderDecorator" << '\n';

}

};

**class** **ScrollDecorator**: **public** Decorator

{

**public**:

*// 6. Optional embellishment*

ScrollDecorator(Widget \*w): Decorator(w){}

*/\*virtual\*/*

**void** draw()

{

*// 7. Delegate to base class and add extra stuff*

Decorator::draw();

cout << " ScrollDecorator" << '\n';

}

};

**int** **main**()

{

*// 8. Client has the responsibility to compose desired configurations*

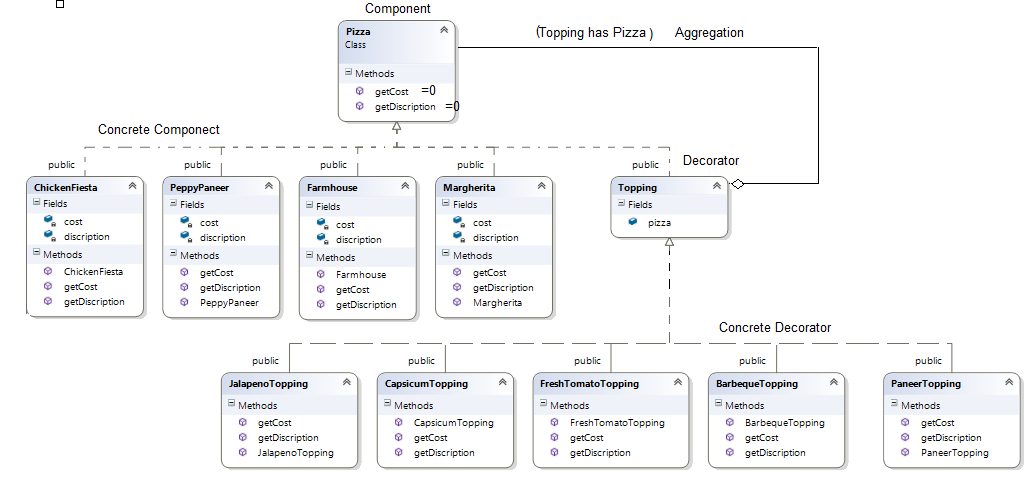
Widget \*aWidget = **new** BorderDecorator(**new** BorderDecorator(**new** ScrollDecorator

(**new** TextField(80, 24))));

aWidget->draw();

}

Example: 2 Pizza Decorator



Relationship between Decorator and component:

<<< ???? >>>

Code:

//Pizza: PeppyPaneer, Farmhouse, Margherita, ChickenFiesta

//Pizza Costs are-

#define PEPPYPANEERCOST 1.10

#define FARMHOUSECOST 2.20

#define MARGHERITACOST 1.75

#define CHICKENFIESTACOST 3.75

//Topping: FreshTomato, Paneer, Jalapeno, Capsicum, Barbeque

//Topping Costs are-

#define FRESHTOMATOCOST 0.20

#define PANNERCOST 0.30

#define JALAPENACOST 0.50

#define CAPSICUMCOST 0.10

#define BARBEQUECOST 0.25

class Pizza {

public:

virtual string getDiscription() = 0;

virtual double getCost() = 0;

};

class PeppyPaneer :public Pizza {

string discription;

double cost;

public:

PeppyPaneer() {

discription = "PeppyPaneerPizza";

cost = PEPPYPANEERCOST;

}

string getDiscription() {

return discription;

}

double getCost() {

return cost;

}

};

class Farmhouse : public Pizza {

string discription;

double cost;

public:

Farmhouse() {

discription = "FarmhousePizza";

cost = FARMHOUSECOST;

}

string getDiscription() {

return discription;

}

double getCost() {

return cost;

}

};

class Margherita :public Pizza {

string discription;

double cost;

public:

Margherita() {

discription = "MargheritaPizza";

cost = MARGHERITACOST;

}

string getDiscription() {

return discription;

}

double getCost() {

return cost;

}

};

class ChickenFiesta :public Pizza {

string discription;

double cost;

public:

ChickenFiesta() {

discription = "ChickenFiestaPizza";

cost = CHICKENFIESTACOST;

}

string getDiscription() {

return discription;

}

double getCost() {

return cost;

}

};

class Topping:public Pizza {

public:

Pizza \*pizza;

};

class FreshTomatoTopping :public Topping {

public:

FreshTomatoTopping(Pizza \*pizza) {

this->pizza = pizza;

}

string getDiscription() {

return pizza->getDiscription() + " With FreshTomatoTopping";

}

double getCost() {

return pizza->getCost() + FRESHTOMATOCOST;

}

};

class PaneerTopping :public Topping {

public:

PaneerTopping(Pizza \*pizza) {

this->pizza = pizza;

}

string getDiscription() {

return pizza->getDiscription() + " With PaneerTopping";

}

double getCost() {

return pizza->getCost() + PANNERCOST;

}

};

class JalapenoTopping :public Topping {

public:

JalapenoTopping(Pizza \*pizza) {

this->pizza = pizza;

}

string getDiscription() {

return pizza->getDiscription() + " With JalapenoTopping";

}

double getCost() {

return pizza->getCost() + JALAPENACOST;

}

};

class CapsicumTopping :public Topping {

public:

CapsicumTopping(Pizza \*pizza) {

this->pizza = pizza;

}

string getDiscription() {

return pizza->getDiscription() + " With CapsicumTopping";

}

double getCost() {

return pizza->getCost() + CAPSICUMCOST;

}

};

class BarbequeTopping :public Topping {

public:

BarbequeTopping(Pizza \*pizza) {

this->pizza = pizza;

}

string getDiscription() {

return pizza->getDiscription() + " With BarbequeTopping";

}

double getCost() {

return pizza->getCost() + BARBEQUECOST;

}

};

Testing..main()

There is two ways to call in main. Choose any of one...

PeppyPaneer p1;

FreshTomatoTopping t1(&p1);

FreshTomatoTopping t2(&t1);

CapsicumTopping t3(&t2);

std::cout << t3.getDiscription().c\_str() << "::"<< t3.getCost()<<endl;

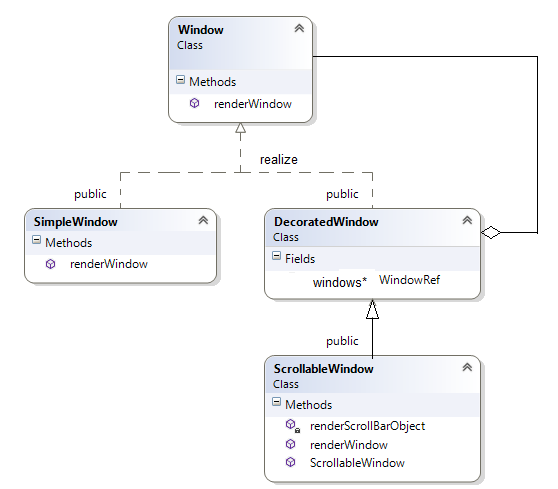
OR,

Pizza \*p1 = new ChickenFiesta();

p1 = new BarbequeTopping(p1);

p1 = new FreshTomatoTopping(p1);

std::cout << p1->getDiscription().c\_str() << "::" << p1->getCost() << endl;

**Decorator Design Pattern - GUI Application:** 

Note: compare with previous example here Decorator class in not abstract, see below implementation.

//Window Interface, Component window

class Window {

public:

virtual void renderWindow() = 0;

};

//Window implementation, Concrete implementation

class SimpleWindow: public Window {

public:

void renderWindow() {

// implementation of rendering details

std::cout << "Simple windows Rendering";

}

};

class DecoratedWindow: public Window {

public:

DecoratedWindow(Window \*ref):WindowRef(ref){}

void renderWindow() {

WindowRef->renderWindow();

}

private:

Window \*WindowRef;

};

//Concrete Decorator with extended state. Scrollable window creates a window that is

//Scrollable

class ScrollableWindow :public DecoratedWindow {

public:

ScrollableWindow(Window \*ref) :DecoratedWindow(ref) {}

void renderWindow() {

// render scroll bar

renderScrollBarObject();

// render decorated window

DecoratedWindow::renderWindow();

}

private:

void renderScrollBarObject() {

prepare scroll bar like.

ScrollBar \*scr= new ScrollBar();

add ScrollBar component into Window, i.e. windows now decorated with scrollbar

WindowRef->addComponect(scr);

std::cout << "Decorated with scrollBar ";

}

};

Test code.

Window \*window = new SimpleWindow();

window->renderWindow();

// at some point later

// maybe text size becomes larger than the window

// thus the scrolling behavior must be added

// decorate old window

std::cout << "\n";

window = new ScrollableWindow(window);

// now window object

// has additional behavior / state

window->renderWindow();