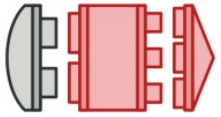
Structural Design Patterns

Structural design patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

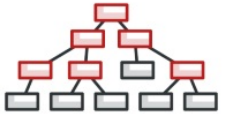


Adapter

Allows objects with incompatible interfaces to collaborate.

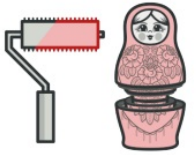
Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.



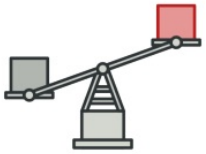
Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Facade

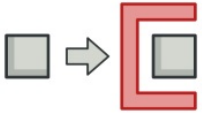
Provides a simplified interface to a library, a framework, or any other complex set of classes.

17th sept



Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.



Proxy

Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

DECORATOR

Also known as: Wrapper

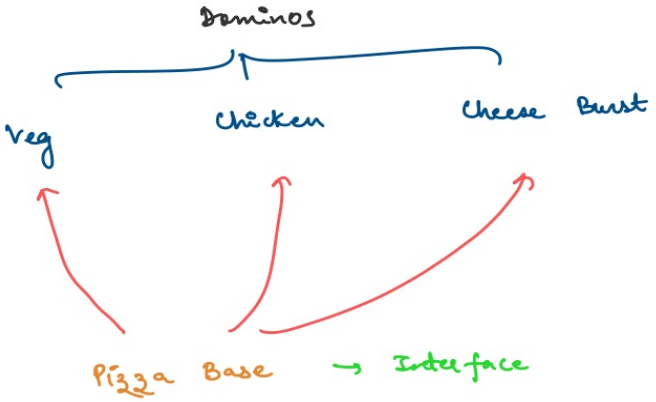
Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

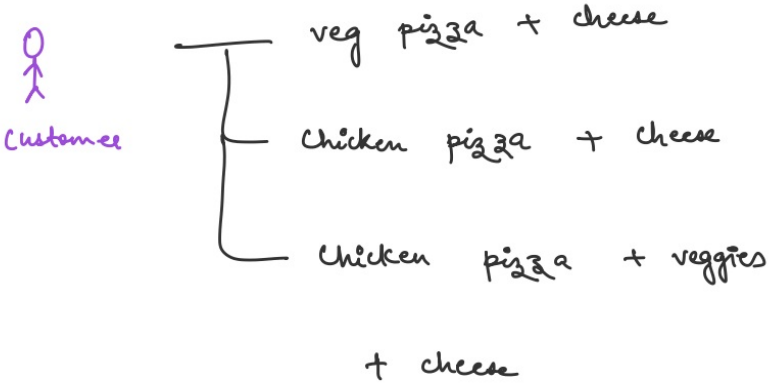
This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

decoration-->functionality

here we are building new variations of previously generated pizzas
so here cant we use builder design pattern
see below explanation



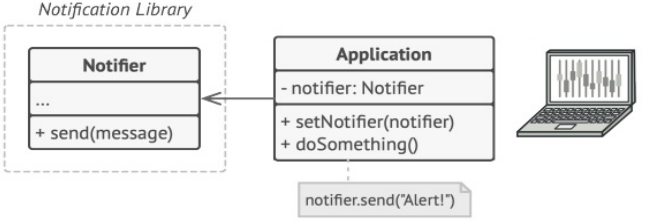
now we want to add some modification like see below



☹ Problem

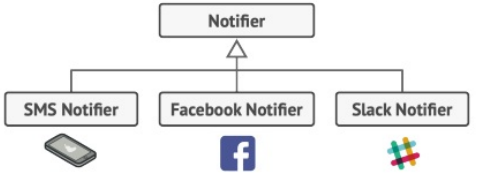
Imagine that you're working on a notification library which lets other programs notify their users about important events.

The initial version of the library was based on the `Notifier` class that had only a few fields, a constructor and a single `send` method. The method could accept a message argument from a client and send the message to a list of emails that were passed to the notifier via its constructor. A third-party app which acted as a client was supposed to create and configure the notifier object once, and then use it each time something important happened.



A program could use the notifier class to send notifications about important events to a predefined set of emails.

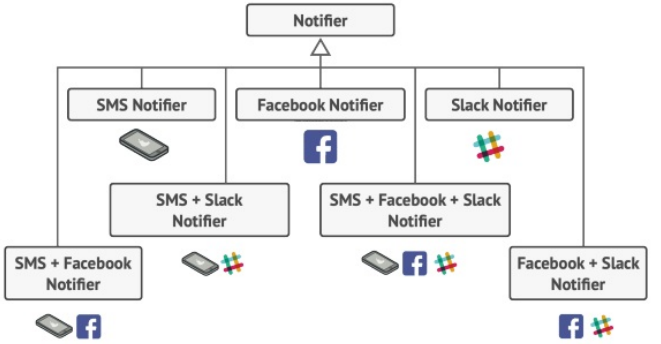
At some point, you realize that users of the library expect more than just email notifications. Many of them would like to receive an SMS about critical issues. Others would like to be notified on Facebook and, of course, the corporate users would love to get Slack notifications.



Each notification type is implemented as a notifier's subclass.

How hard can that be? You extended the `Notifier` class and put the additional notification methods into new subclasses. Now the client was supposed to instantiate the desired notification class and use it for all further notifications.

But then someone reasonably asked you, "Why can't you use several notification types at once? If your house is on fire, you'd probably want to be informed through every channel."



Combinatorial explosion of subclasses.

You tried to address that problem by creating special subclasses which combined several notification methods within one class. However, it quickly became apparent that this approach would bloat the code immensely, not only the library code but the client code as well. You have to find some other way to structure notifications classes so that their number won't accidentally break some Guinness record.

😊 Solution

Extending a class is the first thing that comes to mind when you need to alter an object's behavior. However, inheritance has several serious caveats that you need to be aware of.

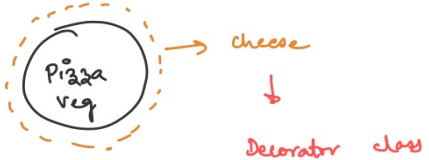
- Inheritance is static. You can't alter the behavior of an existing object at runtime. You can only replace the whole object with another one that's created from a different subclass.
- Subclasses can have just one parent class. In most languages, inheritance doesn't let a class inherit behaviors of multiple classes at the same time.

One of the ways to overcome these caveats is by using *Aggregation* or *Composition*[1] instead of *Inheritance*. Both of the alternatives work almost the same way: one object *has a* reference to another and delegates it some work, whereas with inheritance, the object itself *is* able to do that work, inheriting the behavior from its superclass.

Builder design pattern
↳ objects
↳ we cannot change these objects later
↓
Addressing object complexity at compile time

Decorator design pattern
↳ object
↳ which can be changed based upon the requirement.
↓
Addressing the complexity of object at runtime.
↓
Decorating the object with the new parameters

on builder question interviewer can ask can we use decorator deign pattern and vice versa



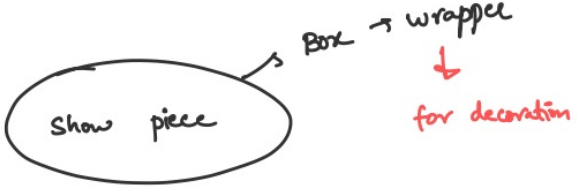
Definition

Decorator design pattern states that we need to attach additional responsibilities to an object dynamically.

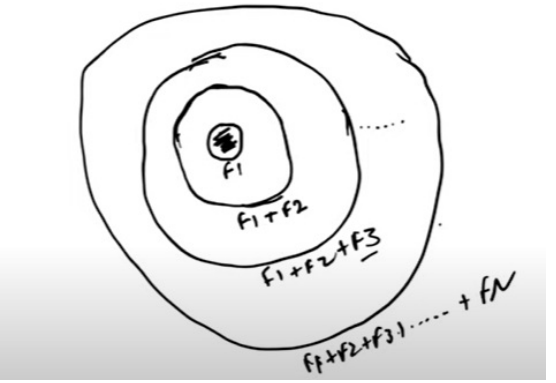
Decorator provides a flexible alternative to sub-classing for extending flexibility.

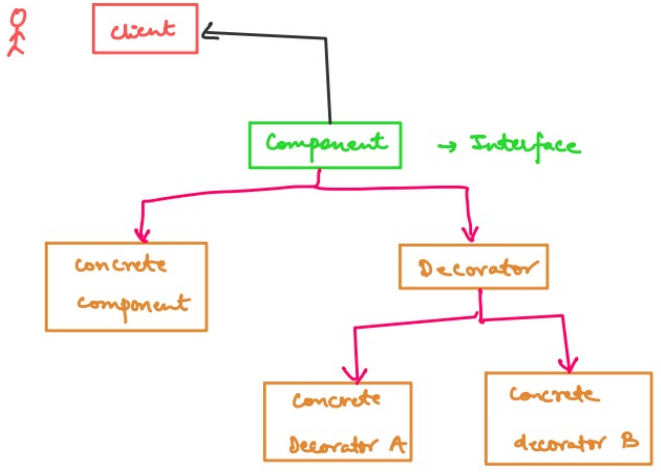
In other words, the decorator pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

Decorator pattern allows a user to add new functionality to an existing object without altering its structure.



This design pattern acts as a wrapper to the existing classes and thats why it is also called as wrapper.





in decorator design pattern we are wrapping the simple object into decorator object like see a gfg example on next page

```java
// Interface named Shape
public interface Shape {

    // Method inside interface
    void draw();
}
```

```java
// Class 1
// Class 1 will be implementing the Shape interface

// Rectangle.java
public class Rectangle implements Shape {

    // Overriding the method
    @Override public void draw()
    {
        // /Print statement to execute when
        // draw() method of this class is called
        // later on in the main() method
        System.out.println("Shape: Rectangle");
    }
}
```

```java
// Class 2
// Abstract class
// ShapeDecorator.java
public abstract class ShapeDecorator implements Shape {

    // Protected variable
    protected Shape decoratedShape;

    // Method 1
    // Abstract class method
    public ShapeDecorator(Shape decoratedShape)
    {
        // This keywordd refers to current object itself
        this.decoratedShape = decoratedShape;
    }

    // Method 2 - draw()
    // Outside abstract class
    public void draw() { decoratedShape.draw(); }
}
```

```java
// Class 3
// Concrete class extending the abstract class
// RedShapeDecorator.java
public class RedShapeDecorator extends ShapeDecorator {

    public RedShapeDecorator(Shape decoratedShape)
    {
        super(decoratedShape);
    }

    @Override public void draw()
    {
        decoratedShape.draw();
        setRedBorder(decoratedShape);
    }

    private void setRedBorder(Shape decoratedShape)
    {
      // Display message whenever function is called
        System.out.println("Border Color: Red");
    }
}
```

Output:

```
Circle with normal border
Shape: Circle

Circle of red border
Shape: Circle
Border Color: Red

Rectangle of red border
Shape: Rectangle
Border Color: Red
```

```java
// Main driver method
public static void main(String[] args)
{
    // Creating an object of Shape interface
    // inside the main() method
    Shape circle = new Circle();

    Shape redCircle
        = new RedShapeDecorator(new Circle());

    Shape redRectangle
        = new RedShapeDecorator(new Rectangle());

    // Display message
    System.out.println("Circle with normal border");

    // Calling the draw method over the
    // object calls as created in
    // above classes

    // Call 1
    circle.draw();

    // Display message
    System.out.println("\nCircle of red border");

    // Call 2
    redCircle.draw();

    // Display message
    System.out.println("\nRectangle of red border");

    // Call 3
    redRectangle.draw();
}
```

see RedShapeDecorator is wrapping up the Shape object and each Decorator is accepting shape in constructor and each Decorator is going implementing shape only so we can wrap one decoartor over other so this is also called as wrapper design pattern.
we are not chnging the existing shape class

Patterns > Decorator > Icar.java > Icar

```java
package Patterns.Decorator;

public interface Icar {
    public void manufactureCar();
}

```

Patterns > Decorator > BasicCar.java > BasicCar > manufactureCar()

```java
package Patterns.Decorator;

public class BasicCar implements Icar {

    @Override
    public void manufactureCar() {
        System.out.println("Manufacture of basic Car");
    }

}

```

Patterns > Decorator > CarDecorator.java > CarDecorator > manu

```java
package Patterns.Decorator;

public class CarDecorator implements Icar {

    protected Icar car;

    CarDecorator(Icar car){
        this.car=car;
    }

    @Override
    public void manufactureCar() {
        this.car.manufactureCar();
    }

}

```

Patterns > Decorator > ElectricCarDecorator.java > ...

```java
package Patterns.Decorator;

public class ElectricCarDecorator extends CarDecorator{

    ElectricCarDecorator(Icar car) {
        super(car);
    }

    @Override
    public void manufactureCar() {
        super.manufactureCar();
        System.out.println("Including feature of electric car");
    }

}

```

Patterns > Decorator > SportsCarDecorator.java > SportsCarDecorator > manufactu

```java
package Patterns.Decorator;

public class SportsCarDecorator extends CarDecorator {

    SportsCarDecorator(Icar car) {
        super(car);
    }

    @Override
    public void manufactureCar() {
        this.car.manufactureCar();
        System.out.println("adding feature of of sports car");
    }

}

```

Patterns > Decorator > LuxuryCar.java > LuxuryCar > manufactureCar()

```java
package Patterns.Decorator;

public class LuxuryCar extends CarDecorator{

    LuxuryCar(Icar car) {
        super(car);
    }

    @Override
    public void manufactureCar(){
        this.car.manufactureCar();
        System.out.println("Adding feature of luxury car");
    }

}

```

see various car decorator extending icardecorator which implements icar

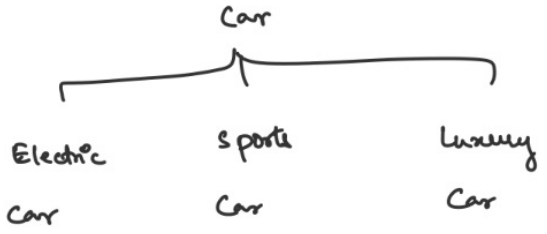
Patterns > Decorator > Application.java > Application > main(String[])

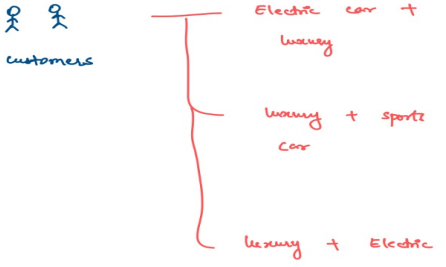
```java
package Patterns.Decorator;

public class Application {
    Run | Debug
    public static void main(String[] args) {
        Icar sportscar=new SportsCarDecorator(new BasicCar());
        sportscar.manufactureCar();
        System.out.println();

        Icar luxuryelectricCar=new LuxuryCar(new ElectricCarDecorator(new BasicCar()));
        luxuryelectricCar.manufactureCar();
    }

}
```

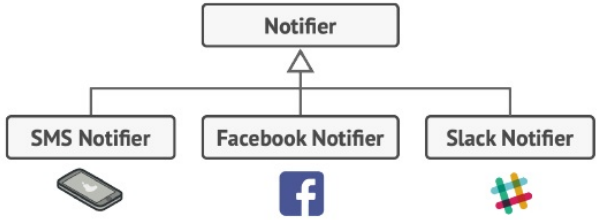
output

```
User/workspaceStorage/4ba69438/redhat.
Manufacture of basic Car
adding feature of of sports car

Manufacture of basic Car
Including feature of electric car
Adding feature of luxury car
```

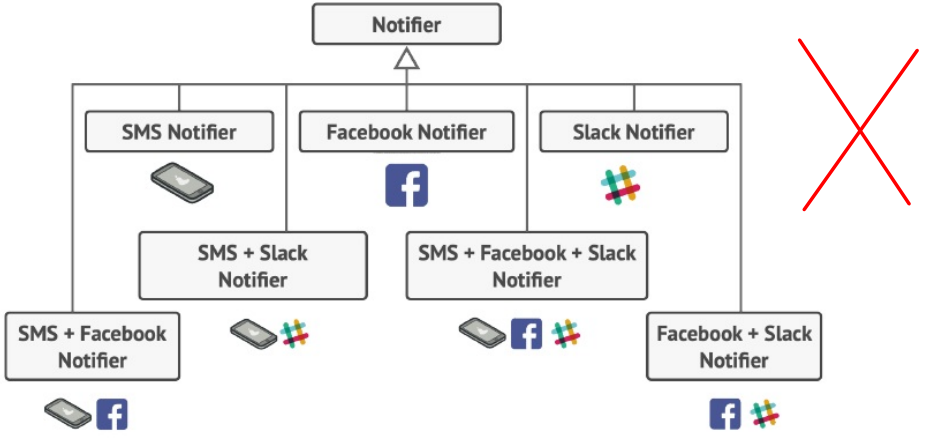
decorator is very commonly asked design pattern
in decorator we can extend more and more decorators w/o changing actual base object code





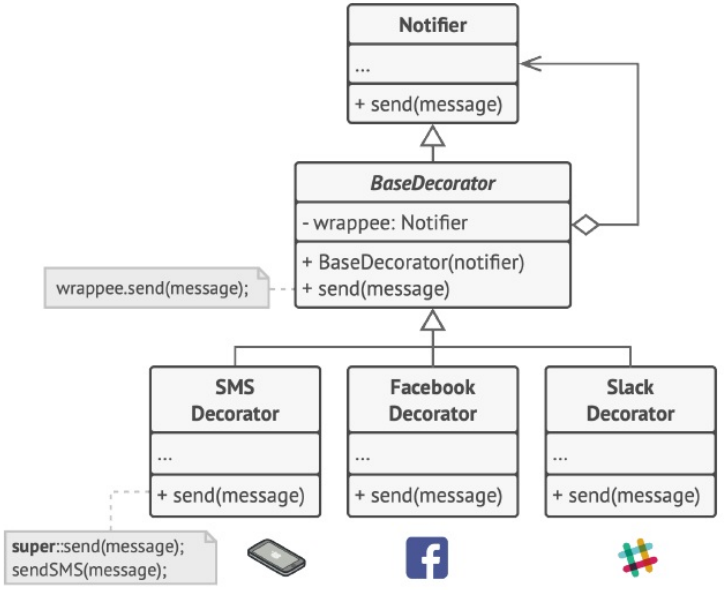


Each notification type is implemented as a notifier's subclass.



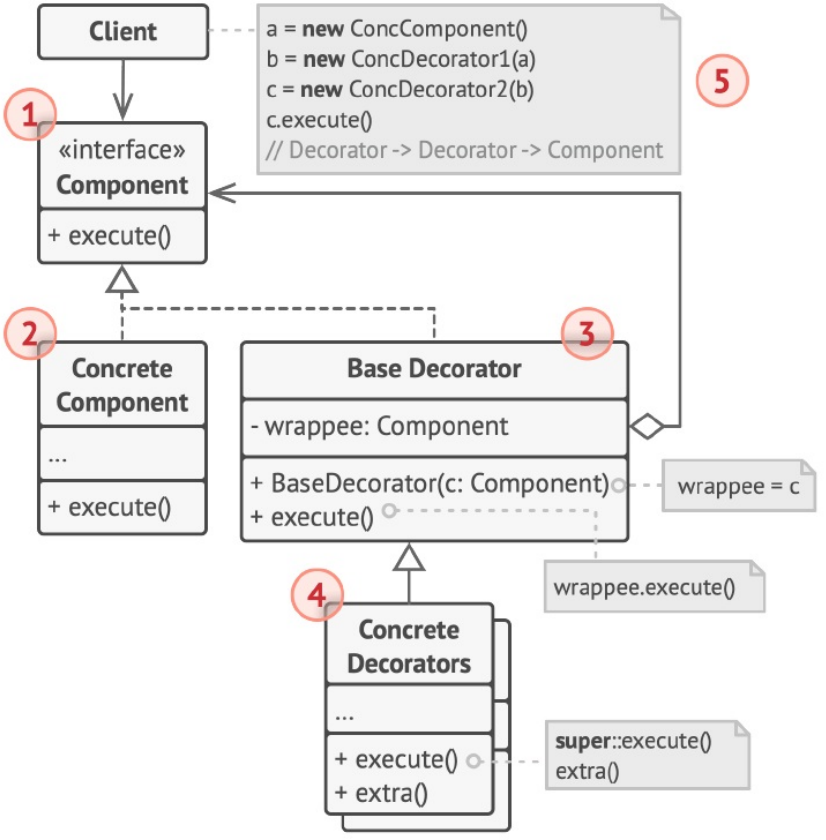
- Inheritance is static. You can't alter the behavior of an existing object at runtime. You can only replace the whole object with another one that's created from a different subclass.
- Subclasses can have just one parent class. In most languages, inheritance doesn't let a class inherit behaviors of multiple classes at the same time.

One of the ways to overcome these caveats is by using *Aggregation* or *Composition*[1] instead of *Inheritance*. Both of the alternatives work almost the same way: one object *has a* reference to another and delegates it some work, whereas with inheritance, the object itself *is* able to do that work, inheriting the behavior from its superclass.



Various notification methods become decorators.

we can extend decorators without changing the base notifier class but if use builder then there will be change in notifier



Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.

The Decorator lets you structure your business logic into layers, create a decorator for each layer and compose objects with various combinations of this logic at runtime. The client code can treat all these objects in the same way, since they all follow a common interface.

Use the pattern when it's awkward or not possible to extend an object's behavior using inheritance.

Many programming languages have the `final` keyword that can be used to prevent further extension of a class. For a final class, the only way to reuse the existing behavior would be to wrap the class with your own wrapper, using the Decorator pattern.

Use the Builder pattern to get rid of a "telescoping constructor".

Say you have a constructor with ten optional parameters. Calling such a beast is very inconvenient; therefore, you overload the constructor and create several shorter versions with fewer parameters. These constructors still refer to the main one, passing some default values into any omitted parameters.

Use the Builder pattern when you want your code to be able to create different representations of some product (for example, stone and wooden houses).

The Builder pattern can be applied when construction of various representations of the product involves similar steps that differ only in the details.

The base builder interface defines all possible construction steps, and concrete builders implement these steps to construct particular representations of the product. Meanwhile, the director class guides the order of construction.

Pros and Cons

- ✓ You can extend an object's behavior without making a new subclass.
- ✓ You can add or remove responsibilities from an object at runtime.
- ✓ You can combine several behaviors by wrapping an object into multiple decorators.
- ✓ *Single Responsibility Principle*. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.

- ✗ It's hard to remove a specific wrapper from the wrappers stack.
- ✗ It's hard to implement a decorator in such a way that its behavior doesn't depend on the order in the decorators stack.
- ✗ The initial configuration code of layers might look pretty ugly.

in builder design pattern object abii built nii hua hai tb usme dalte features but after completion of that ki ye features hai now you cant extend features ki aur features daal de

in decorator as feature alag se class hoti usko sare feature extend krte toh vha pe hm aur nye features daal skte aur yha pe phle object bnta fir feature daalte at runtime

Builder mai phle feature daalte(at compile time) then object bnta hai

So, one should use builder if he wants to limit the object creation with certain properties/features. For example there are 4-5 attributes which are mandatory to be set before the object is created or we want to freeze object creation until certain attributes are not set yet. Basically, use it instead of constructor – as Joshua Bloch states in Effective Java, 2nd Edition. The builder exposes the attributes the generated object should have, but hides how to set them.

Decorator is used to add new features of an existing object to create a new object. There is no restriction of freezing the object until all its features are added.
Both are using composition so they might look similar but they differ largely in their use case and intention.

to remember ki builder mai object bna ni tb feature daalte remember builder is used to avoid telescopic constructor .so jaise constructor ke end mai object bnta vaise hi builder ke end mai object bnta hai

aur decorator ko object milta hai voh use wrap krke nye box(object) mai daal ke de deta