## 2. Decorator Pattern.

Attach additional responsibilities to an object dynamically without modifying its class.

In simple terms:-
- Add new behavior at runtime.
- without changing existing code.
- without using inheritance explosion.

→ wrap an object with another object.
⇒ wrapper implements the same interface.

eg, coffee example.

- Start with plain coffee.
- Add milk.
- Add sugar.
- Add cream.

Structure

1) component Interface.

```
interface Coffee{
  double cost();
}
```

2). Concrete Component.

```
class SimpleCoffee implements Coffee{.
  Public double cost(){ return 50;}
}
```

3) Abstract Decorator

```
abstract class CoffeeDecoreator implements
                                coffee{

  Protecd Coffee coffee;
  CoffeeDecoreator (Coffee coffee){
    this. coffee = coffee;
  }
}
```

4) Concrete Decorator.

```
class Milk Decorator extends CoffeeDecorator{
  Milk Decorator (Coffee coffee){
  super(coffee);
  }
  public double cost (){
    return coffee. cost ()+10;
  }
}
```

```
class SugarDecorator extends CoffeeDecorator
  Sugar Decorator (Coffee coffee){
    super( coffee);
  }
  public double cost(){
    return Coffee.cost()+5;
  }
}
```

```
⇒ Coffee coffee = new SugarDecorator( new MilkDecorator(
                                new SimpleCoffee()));
S.O.Pln ( coffee. cost ()); //65.
```