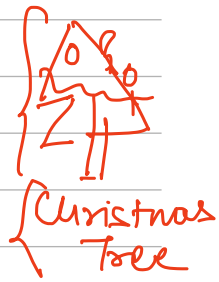
Agenda

Start @ 9.05pm

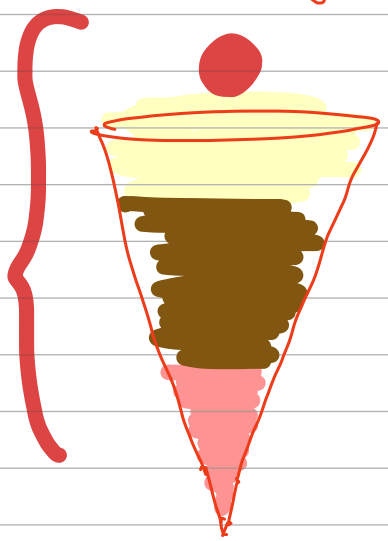
1) Decorator design pattern → IceCream | Pizza
2) Flyweight design pattern → PUBG | BGMI

Decorator design pattern



Icecream mangement System

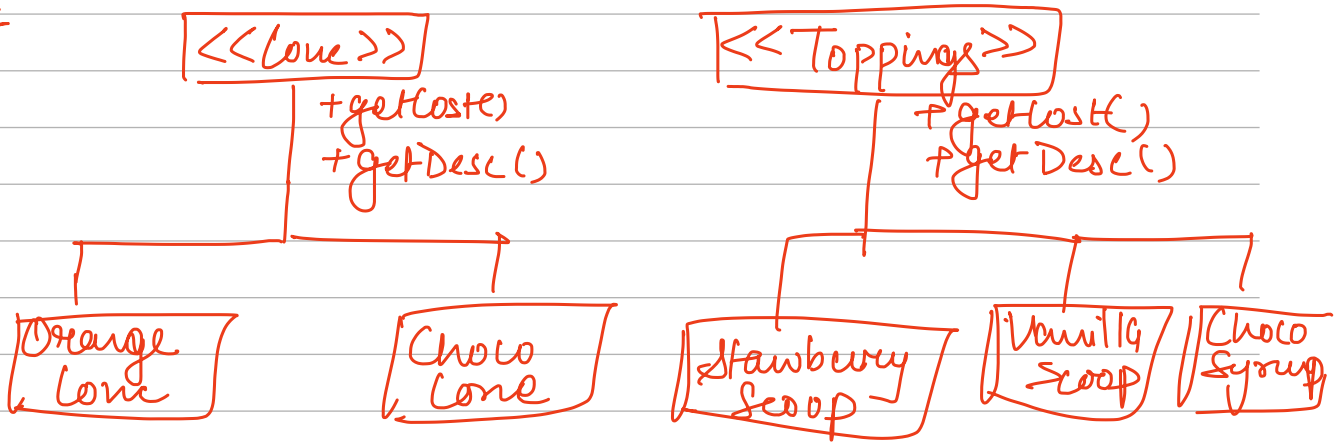
1) App can take orders for icecream
2) Only custom icecreams



→ getCost(): Total cost of icecream
[Cost of Cone + Cost of Stawberry + Cost of Choco + Cost of BS + Cost of Cherry]

→ getDesc(): Orange Cone + Stawberry scoop + Choco Scoop + BS scoop + Cherry

Soln



IceCream

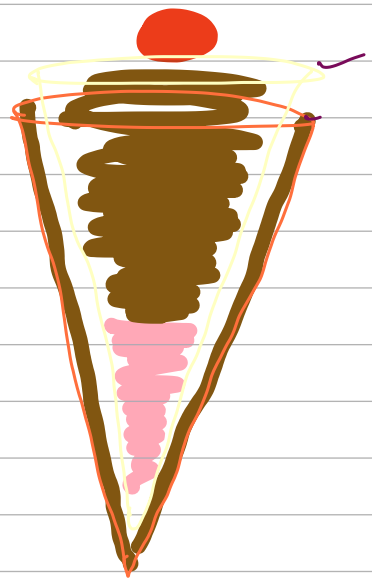
List<Cone>
List<Toppings>

getCost<>() // go through the lists & sum all cost

getDesc() { // go through the lists & append all desc

}

Eg



Order: Orange Cone + Choco Syrup + Vanila Cone + Stawbery scoop + Choco scoop + Cherry

Bill: Orange Cone + Vanila Cone + Choco syrup + Stawbery scoop + Choco scoop + Cherry

---

Soln 2

Single interface : Ingredients

<<Ingredients>>

- ChocoCone
- Orange Cone
- Choco Syrup
- Vanila Scoop
- Cherry
- Choco Chips

ICE Cream

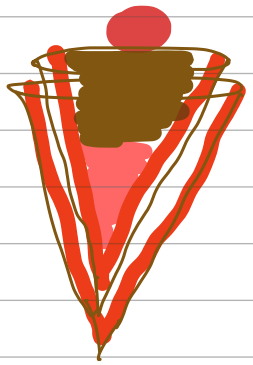
List<Ingredients>

getCost() { ——— }

getDesc() { ——— }

addingd(↓) { ——— }

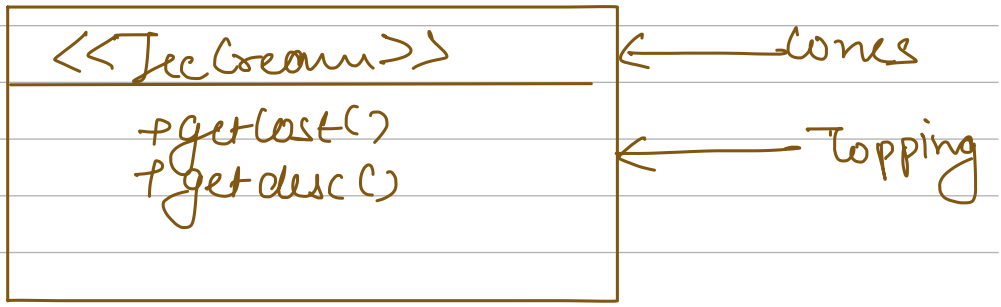
Decorator



⇒ Right from the beginning we had an ice cream and we keep adding the addon on it.

⇒ When we add anything an ice cream, cost and desc will change

Step1 Define an interface / abstract class for the entity we are building



Step2 There are two types of ingredients (Addons)

⇒ For each addon create a class that implements the icecream interface

⇒ Only for base (when start first time)

getCost() = cost of itself

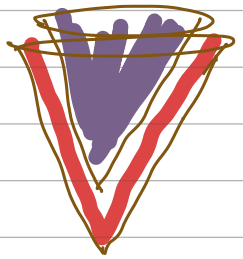
getDesc = desc of itself

for addons

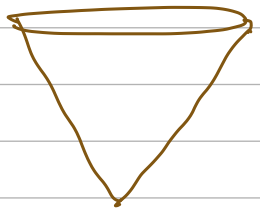
getCost() = Cost of existing icecream + Cost of Addon

getdesc() = desc of existing icecream + desc of addon

eg.

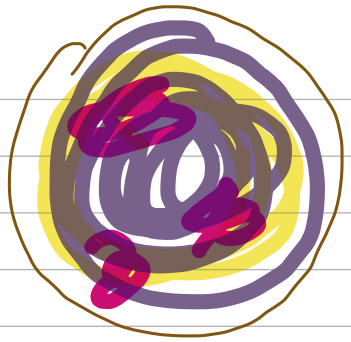


[[[Cost = 10] + 5] + 10] + 15 — 10 + 5 = 15



Cost = 0 + 10

```
Icecream ic = new Cherry(
                new ChocoChips(
                  new ChocoCone(
                    new StrawberrySyrup(
                      new OrangeCone())
                    )
                  )
                )
```



```
OrangeCone
IC ic;
OrangeCone(){
}
OrangeCone(IC ic){
}
getCost{
    if(ic==null)
        return 10;
    else
        return ic.getCost()+10;
}
```

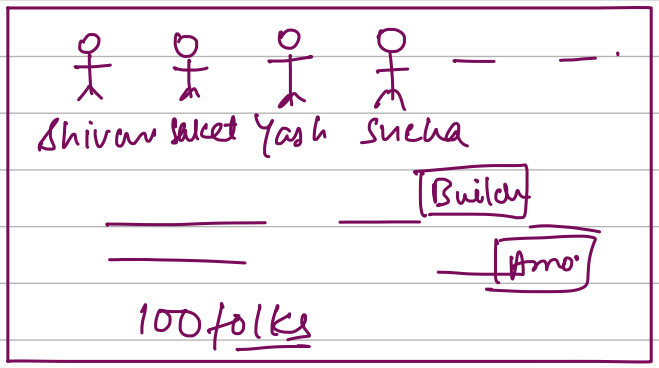
X

```
ChocoSyrup
IC ic;
ChocoSyrup(IC ic)
{
}
```

Flyweight design pattern

Build an online multiplayer

eg PUBG / BGMI / COD / CS



10 Guns

2 guns / player

300 bullets

⇒ Complete state of the game on every player device

2) Every change that happens needs to be transferred to every player

Bullets		
Double : 8B	- radius — SAME	100K
Int : 4B	- color — SAME	
Double : 8B	- weight — SAME	
Int : 4B	- length — SAME	→ Intrinsic
Int : 4B	- range — SAME	
Int : 4B	- max speed — SAME	
Double : 8B	- Max damage — SAME	
3 Double : 24B	- direction X	$x\hat{i}+y\hat{j}+z\hat{k}$
Int : 4B	- curr speed X	} Extrinsic
3 Double : 24B	- target coord X	
byte[] : 1KB	- Image — SAME	
~1.1KB		

1 PUBG = 100 K

Total Mem = 100K x 1.1 KB

≈ 100MB

10 Different type of Bullets

Observation: Even though we have 100K bullets but not all bullets are completely different

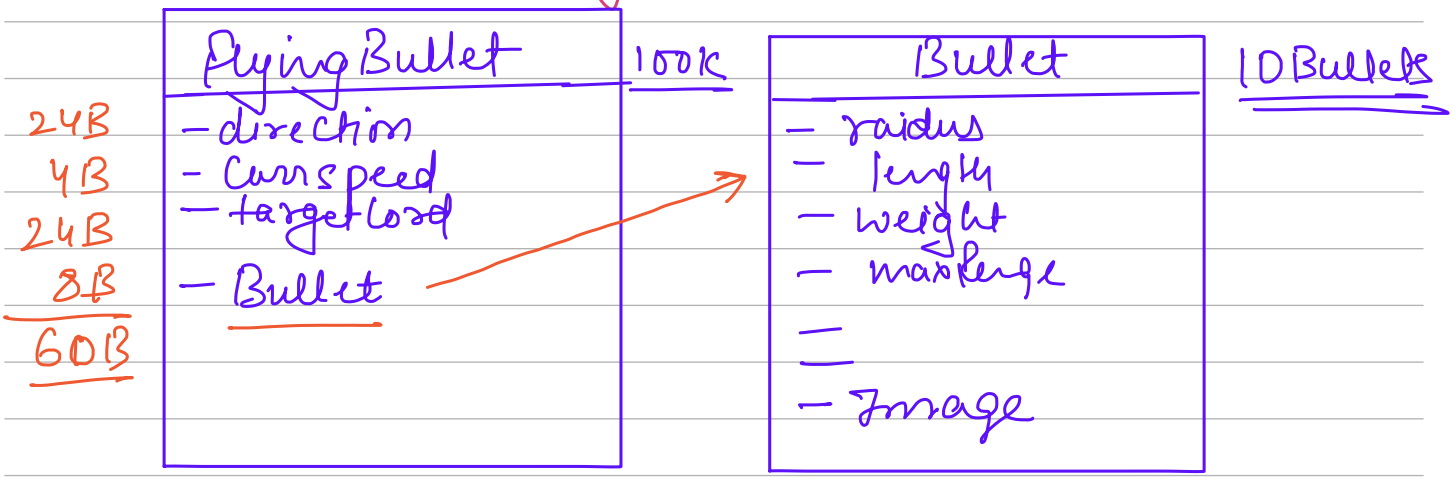
Often we have classes that have 2 types of properties

1) Extensic : Values of which gets changed with time
2) Intrinsic : Values of which don't get change with time

Flyweight : If you have a class where objects have extensic & intrinsic properties.
ALSO if you notice that extra memory is getting utilised due to intrinsic properties then consider flyweight

Divide Class in 2 separate Classes

1) with only intrinsic properties
2) with only extrinsic properties



1 PUBG 100K X 60B = 6MB

10 X 1.1KB ≈ 10KB

≈ 6MB

100MB ⟶ 6MB