⇒ Structural Design Pattern

→ Decorator

→ flyweight

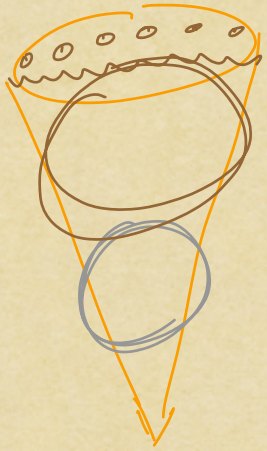
Decorator Design Pattern

Suit of Vadilal Quality Walls

⇒ build an ice-cream ordering system

- Appn should only take order for ice-cream cones
- Customisable
- no changes during making the order

eg ⇒ Orange Cone + Vanilla Scoop + Chocolate Scoop + Choc. Syrup + Choc. chips



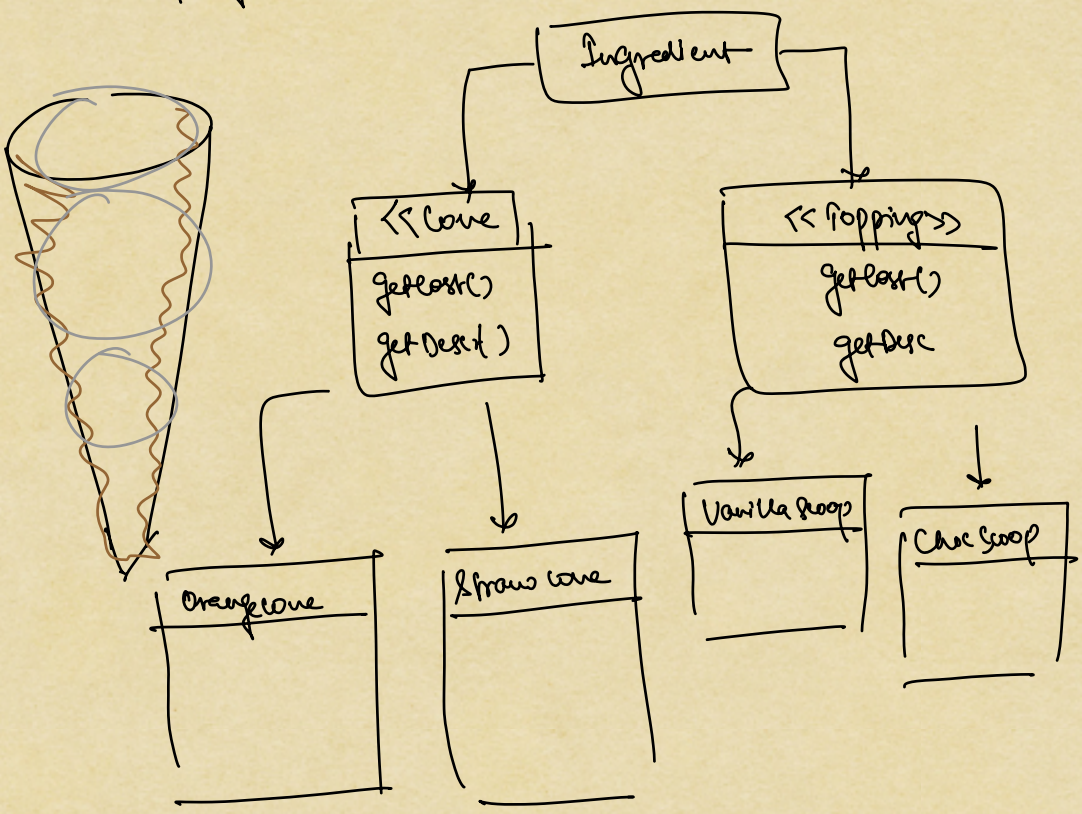
→ build the ice-cream

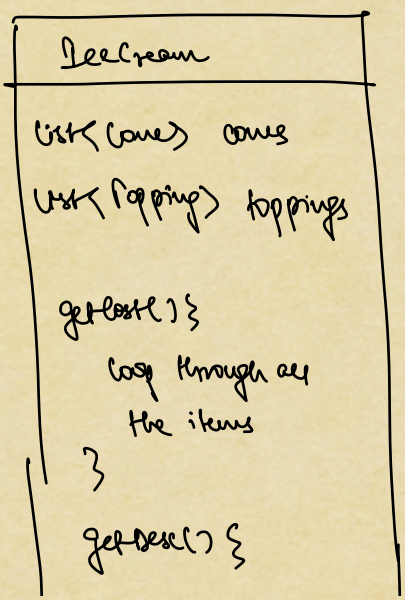
→ cost of the ice-cream

→ description (list of ingredients)

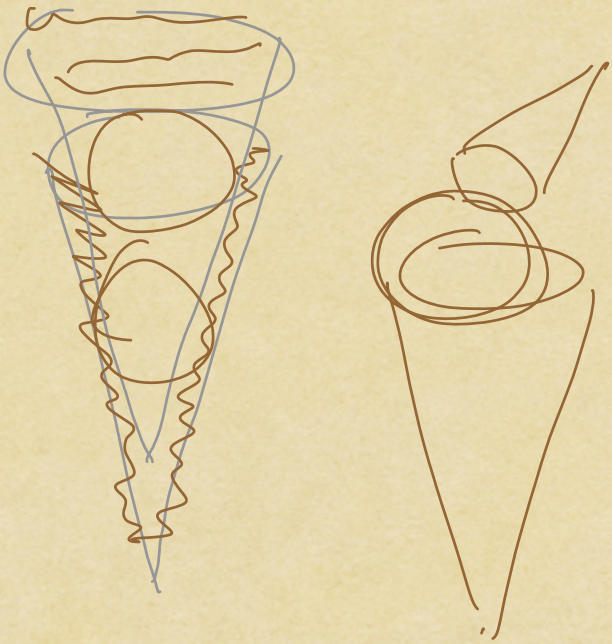
Soln 1

- Cone → base item
- Icecream scoop } → on top of base item
- toppings





loop through
all items
}

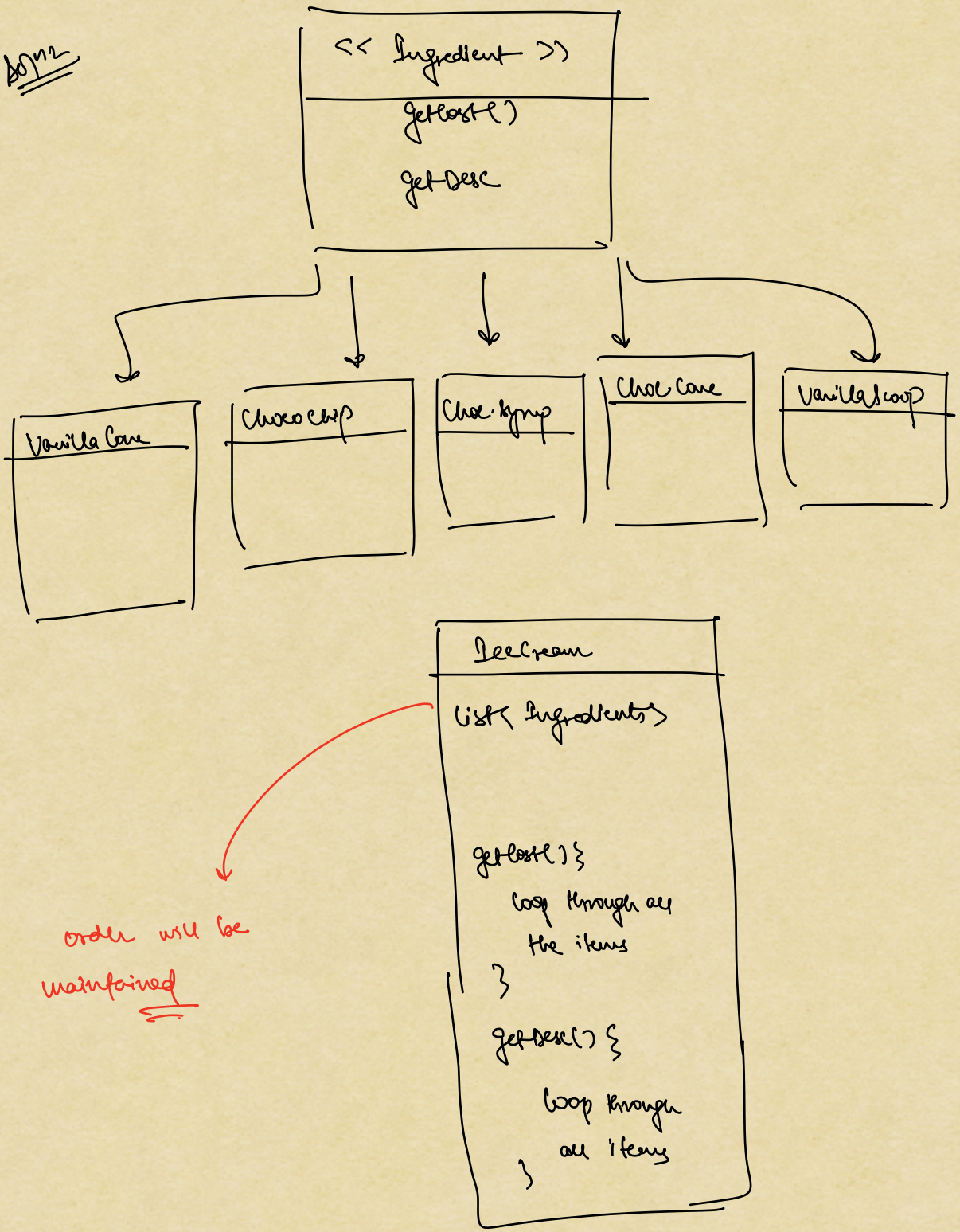


Recipe ⇒ Vanilla Cone + Choco Syrup + Choc Cone + Vanilla Scoop + Choc. Scoop

List(Cones) ⇒ Vanilla Cone, Choc Cone

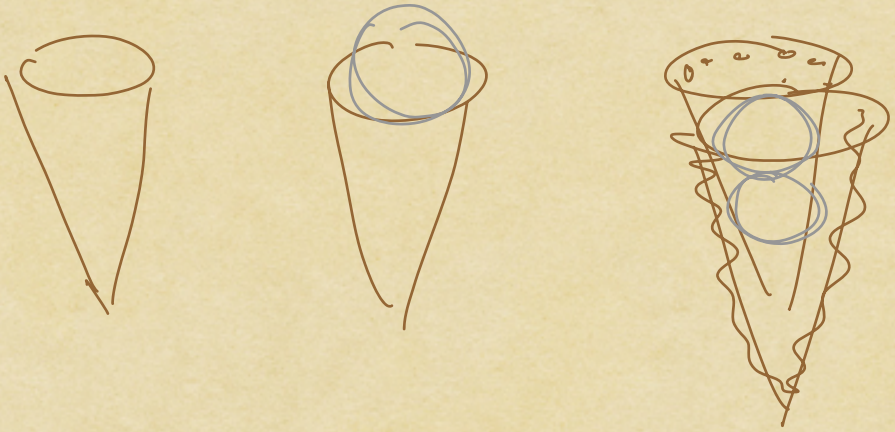
List(toppings) ⇒ Choco Syrup, Vanilla Scoop, Choc. Scoop

Vanilla Cone + Choc Cone + Choco Syrup + Vanilla Scoop + Choc Scoop


<< Ingredient >>
getCost()
getDesc
Vanilla Cone
Choco Chip
Choc Syrup
Choc Cone
Vanilla Scoop
IceCream
List< Ingredients >
getCost() {
Loop through all the items
}
getDesc() {
Loop through all items
}
order will be maintained

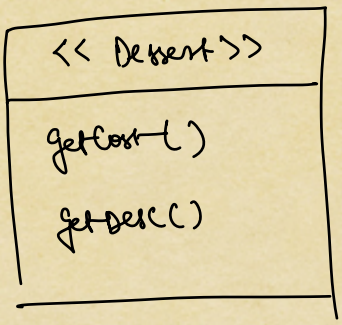
=> Decorator

→ Right from beginning since user starts ordering any set of items can be called as ice-cream.

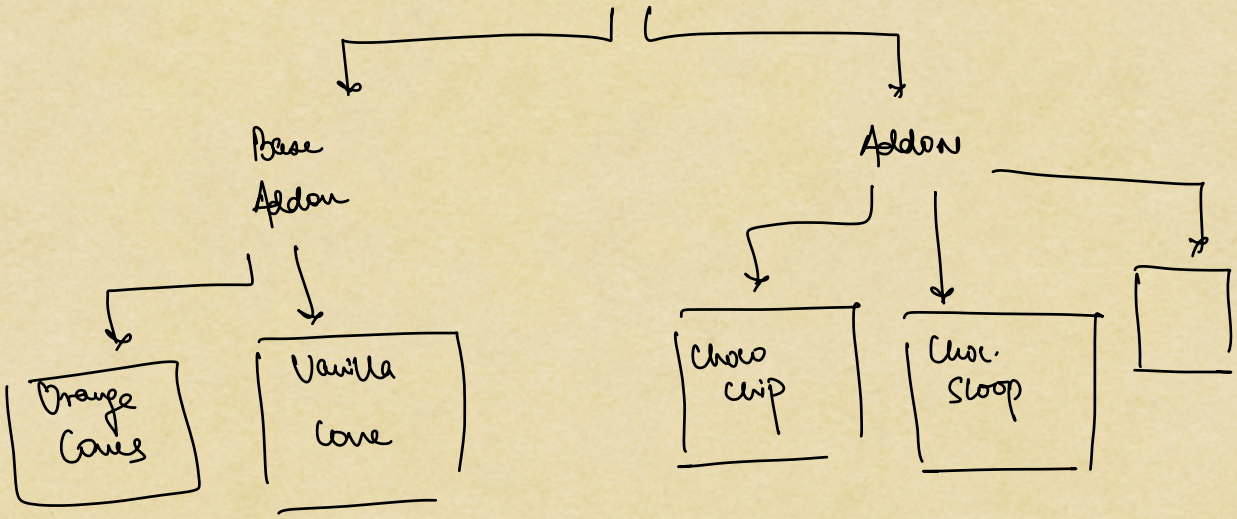


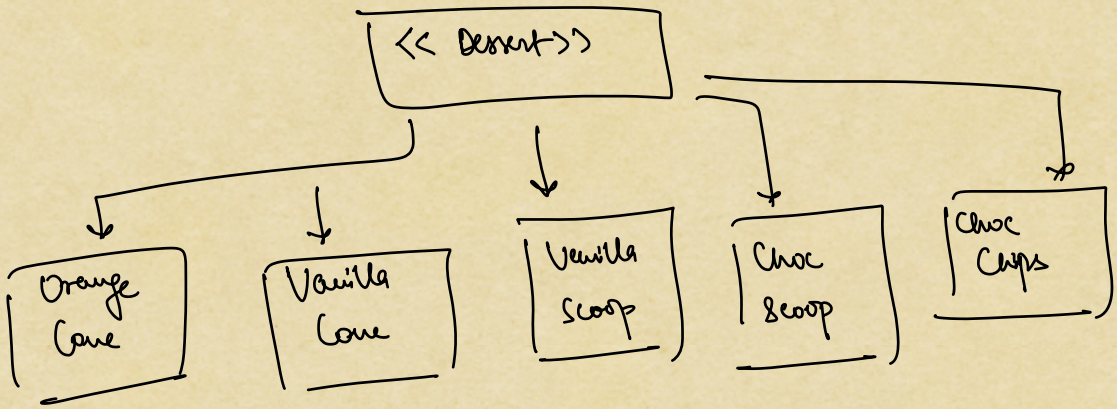
→ Whenever the user adds any item, the cost and desc. of the dessert/ice-cream should change

Step1 => Define an interface/abstract class that represents the thing that we are constructing.



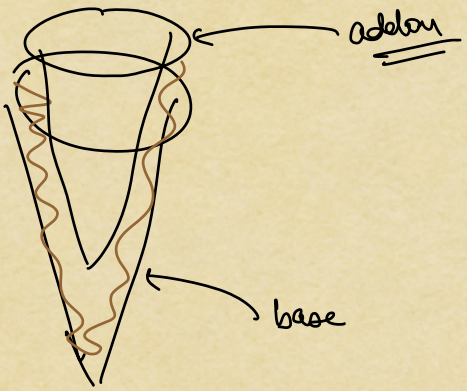
Step 2 There are 2 kinds of items!





⇒ ONLY A BASE ENTITY !

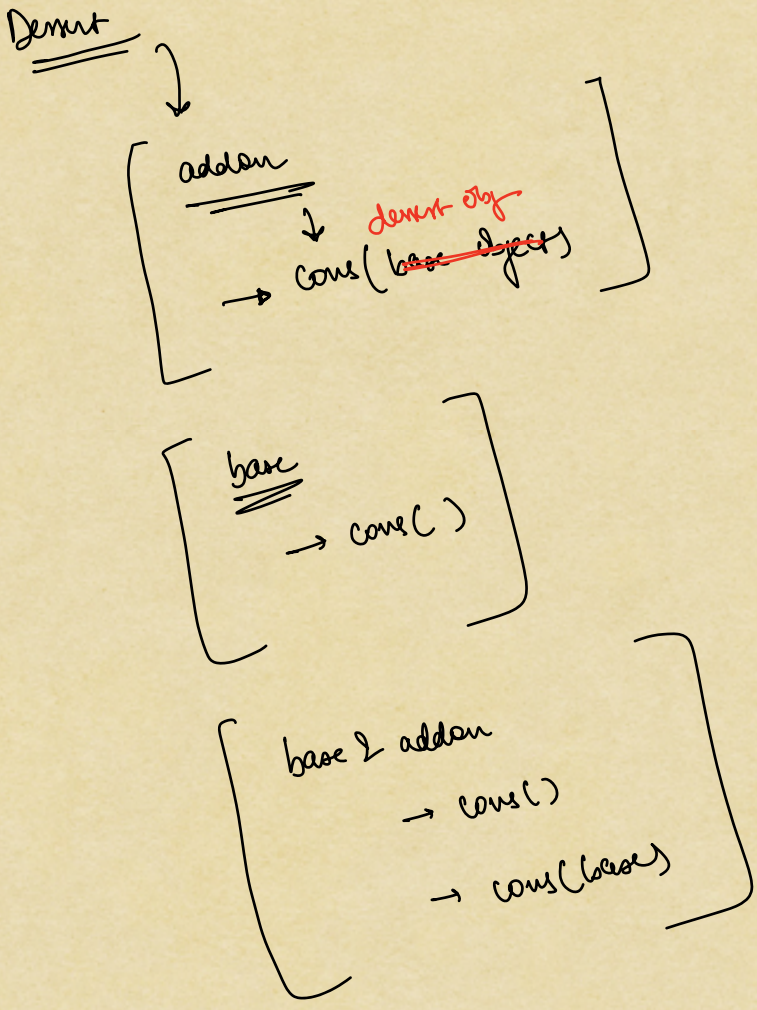
* SO cone is both base and addon



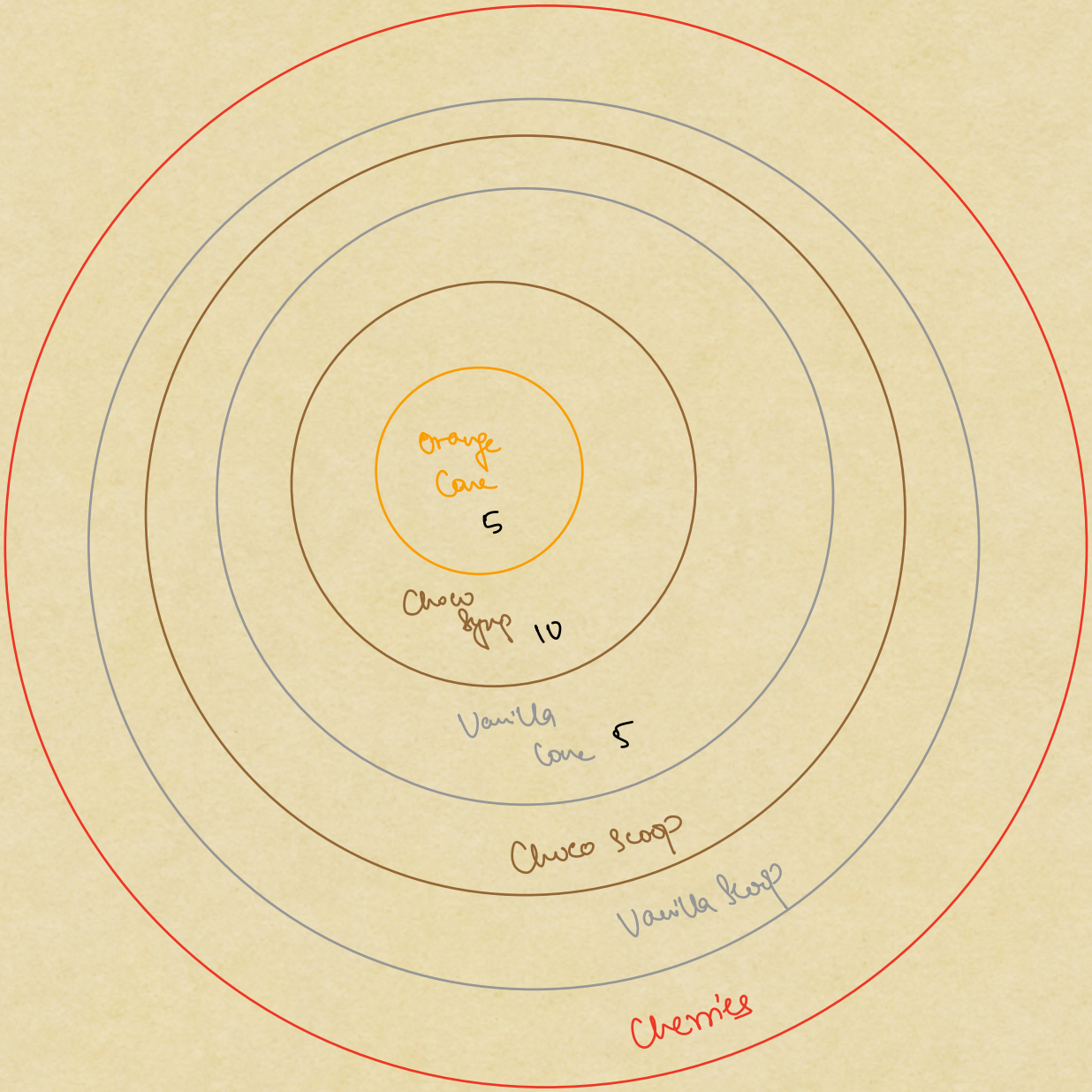
⇒ ONLY AN ADDON

- choco chips
- choco syrup
- choco scoop

⇒ only an addon can't be base entity



⇒ Orange Cone, Choco Syrup, Vanilla Cone, Choc scoop,
Vanilla Scoop, cherry



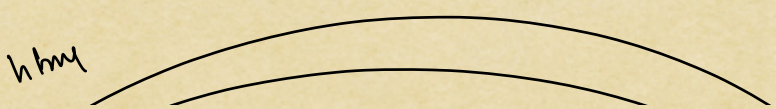
Dessert d =

```
new ChocoScoop(
    new VanillaCone(
        new ChocSyrup(new OrangeCone())
    )
)
```

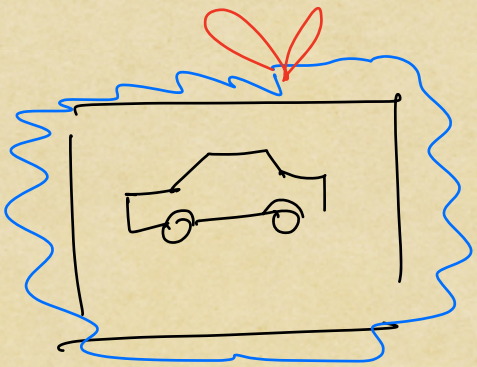
⇒ updating the structure of an object at runtime, can be done via decorator.

HTML

```
<html>
    <body>
        <div>
            <p>
                <h1> Hi </h1>
            <p>
        </div>
    </body>
</html>
```




p
h1
h2



<body>

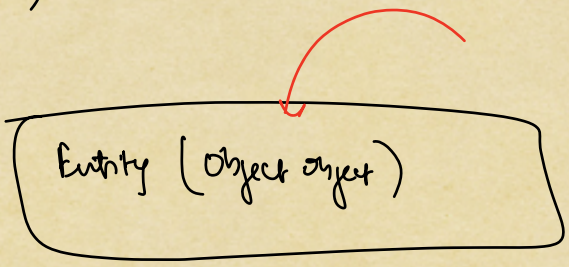
</body>

@Entity @Component

class Student{

}

~~@Entity~~



Spring

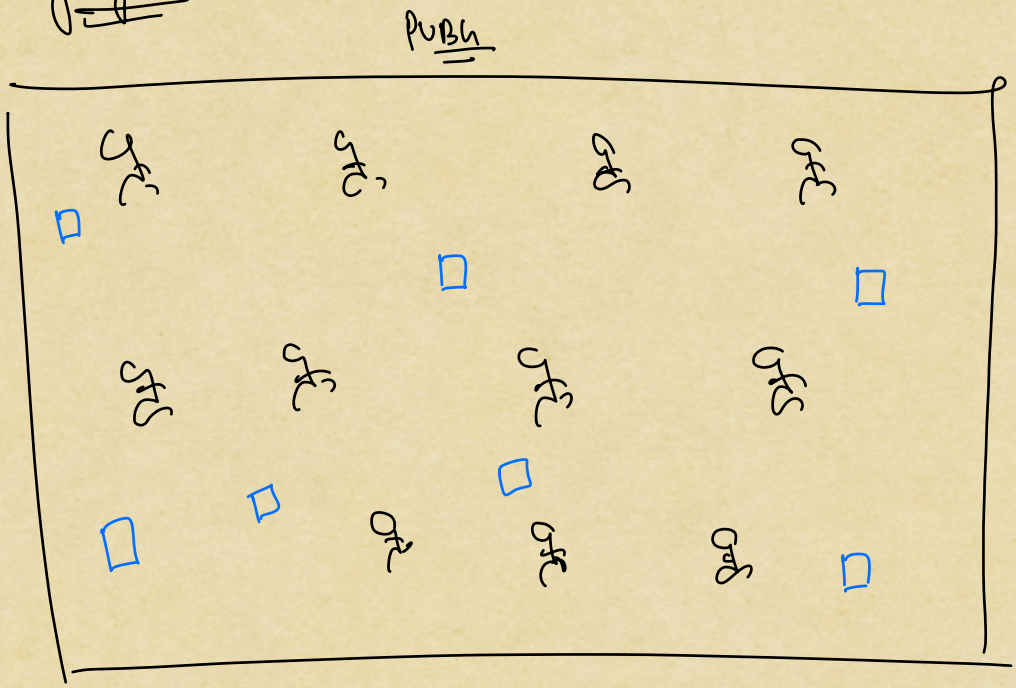
Pizza

base + extra

→ farmhouse

→ cheese burst

⇒ flyweight



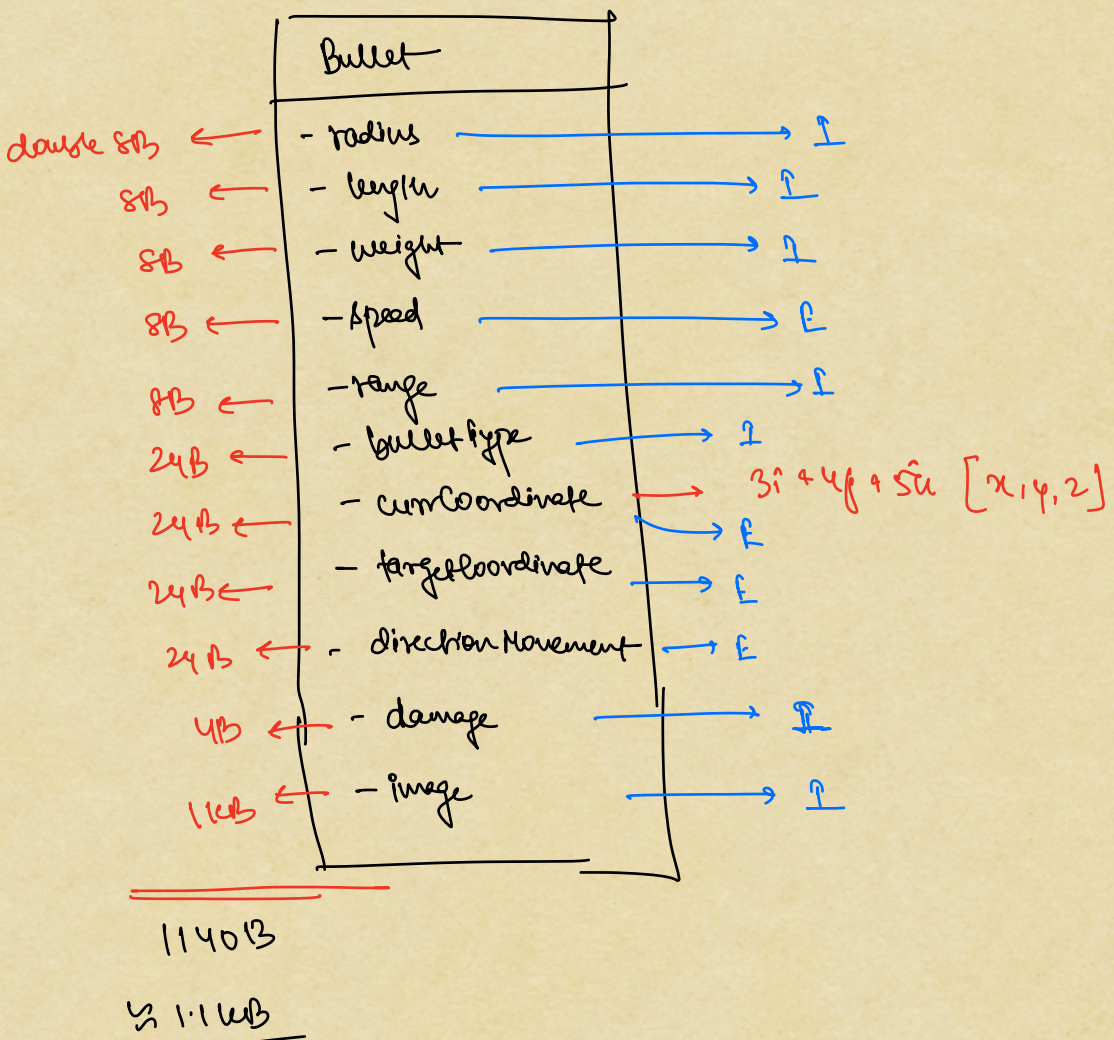
⇒ at the end 1 player remains alive ⇒ WINNER

⇒ 2 guns/player

⇒ 300 bullets / player

→ Complete state of the game is downloaded to the machine of every player

→ Changes of the game are transferred to every machine



Assume: 1 game ⇒ 100000 bullets

memory ⇒ 1.1KB × 100K

≈ 1100 × 100000

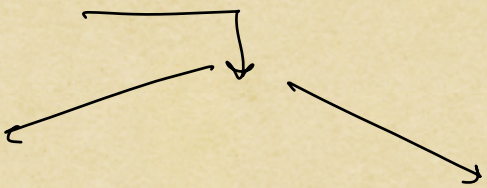
$= 0.11 \times 10^9$

$= 0.11\ GB$

$\approx 110\ MB$

Observation ⇒ even though number of bullet that is very high types of bullets will be very slow ⇒ 10 types assume

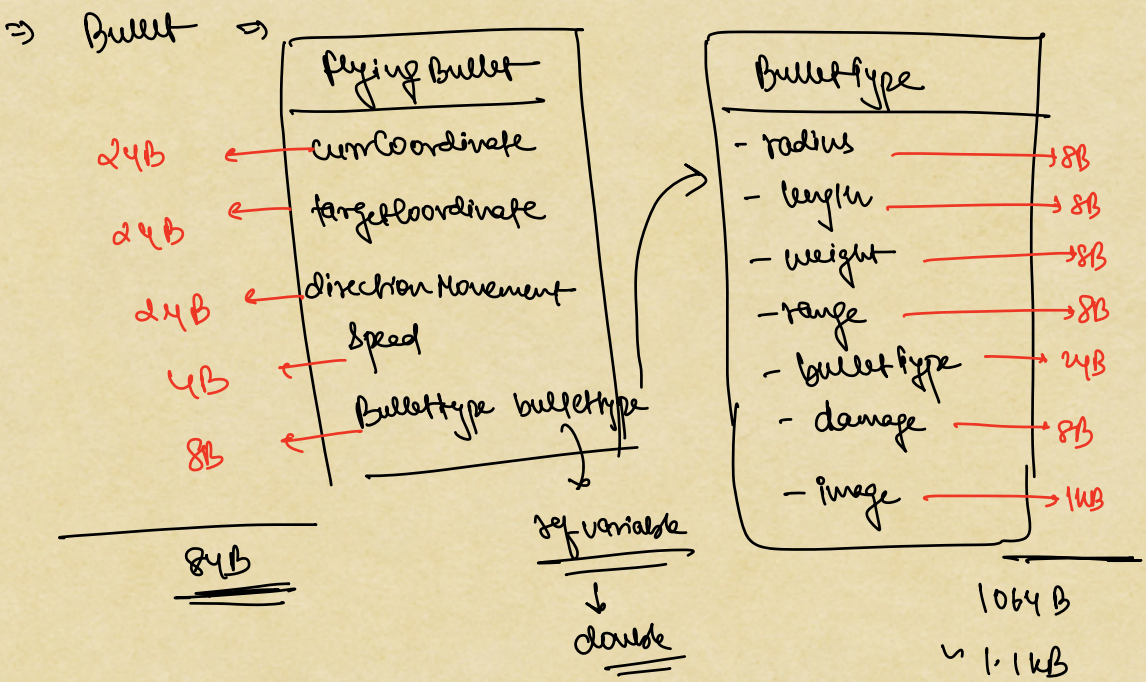
Often times, we have classes that have majority of types of attributes



Extrinsic
[changes per Object]

Intrinsic
[remains same across all/majority objects]

⇒ flyweight ⇒ says keep both separate and reuse intrinsic attributes to achieve flyweight.



10 types ⇒ 1.1 KB × 10 ⇒ 11 KB

100K bullets ⇒ 84B × 100 K

= 8400000

= 84000 KB

= 8.4 MB

Total ⇒ 8.4 MB + 11 KB

≈ 8.5 MB

110MB ⟶ 8.5MB

Student

School

class School {

name

S1

S2

S3

School
=@123

name=Abc

POR

@123

@345

Flying Bullet

BulletType

BulletType

name=

f1

f2

name=Rupa

Super

@123

@345