Agenda

1. Decorator Design Pattern
   - → Ice Cream, Coffee, Pizza
   - → HTML, Spring App^n
2. Flyweight Design Pattern
   - → PubG

→ Interview Q^n

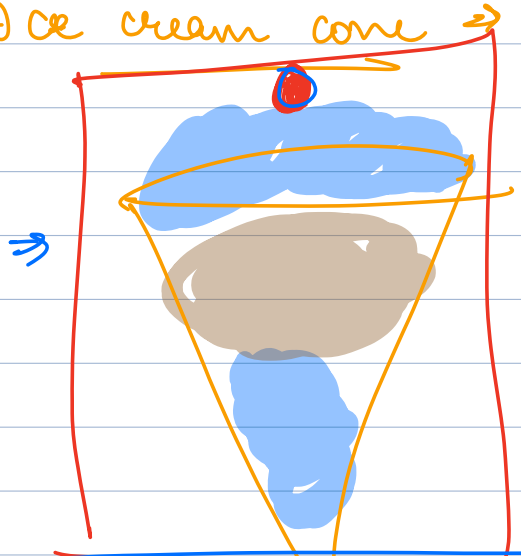
Structural Design Patterns

DECORATOR DESIGN PATTERN

SWE at Vadilal / Kwality Walls

Ice Cream Ordering System

→ Ice cream cone ⇒ Custom Made



⇒ App^n that takes orders for ice cream cones

⇒ Custom Config

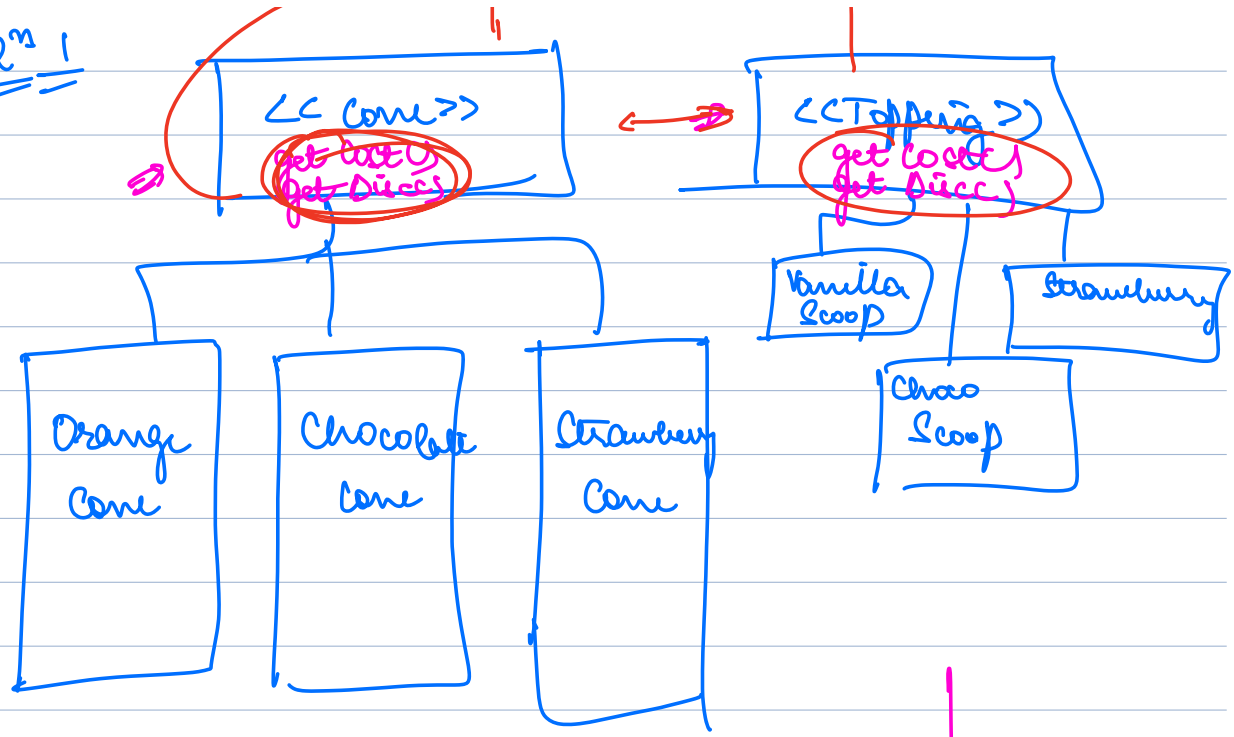
Orange Cone + Vanilla + Choco + Vanilla + Strawberry

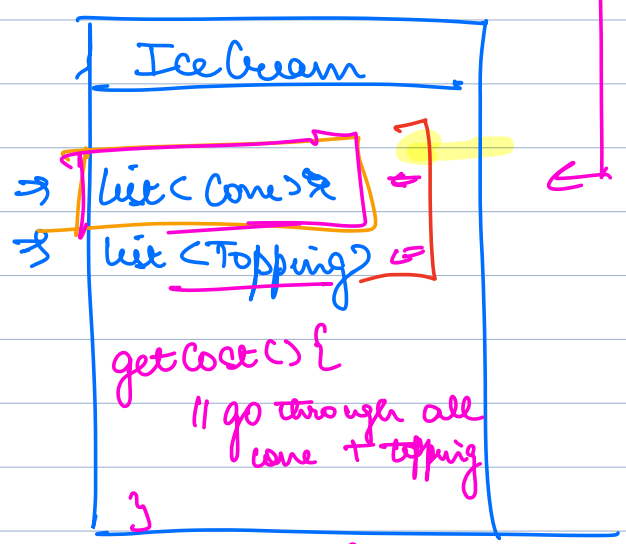
1. Build an ice cream
2. Cost of the ice cream
3. Description (list of ingredients) =

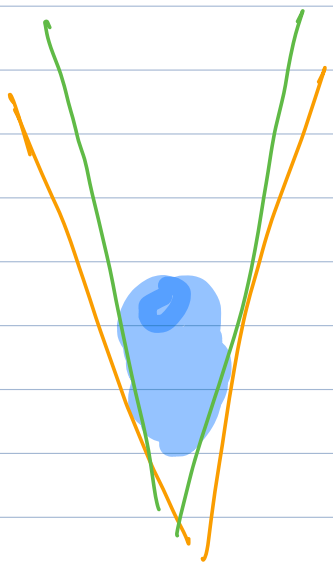
How will you store info about ice cream.

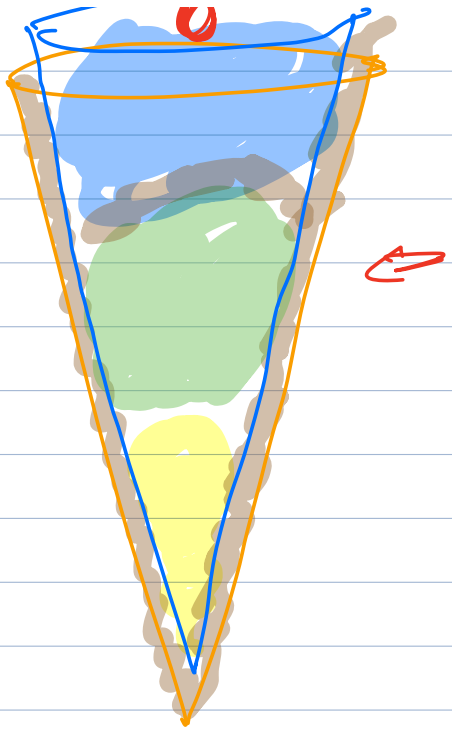
Ingredient

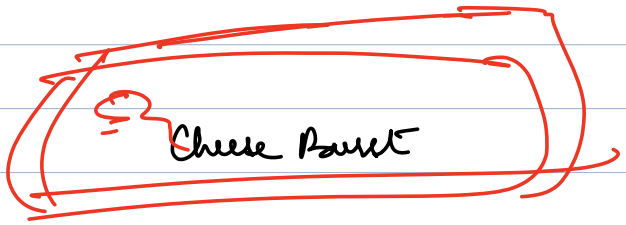
Soln 1


<< Cone >>
getCost()
getDisc()
<< Topping >>
getCost()
getDisc()
Vanilla Scoop
Strawberry
Choco Scoop
Orange Cone
Chocolate Cone
Strawberry Cone

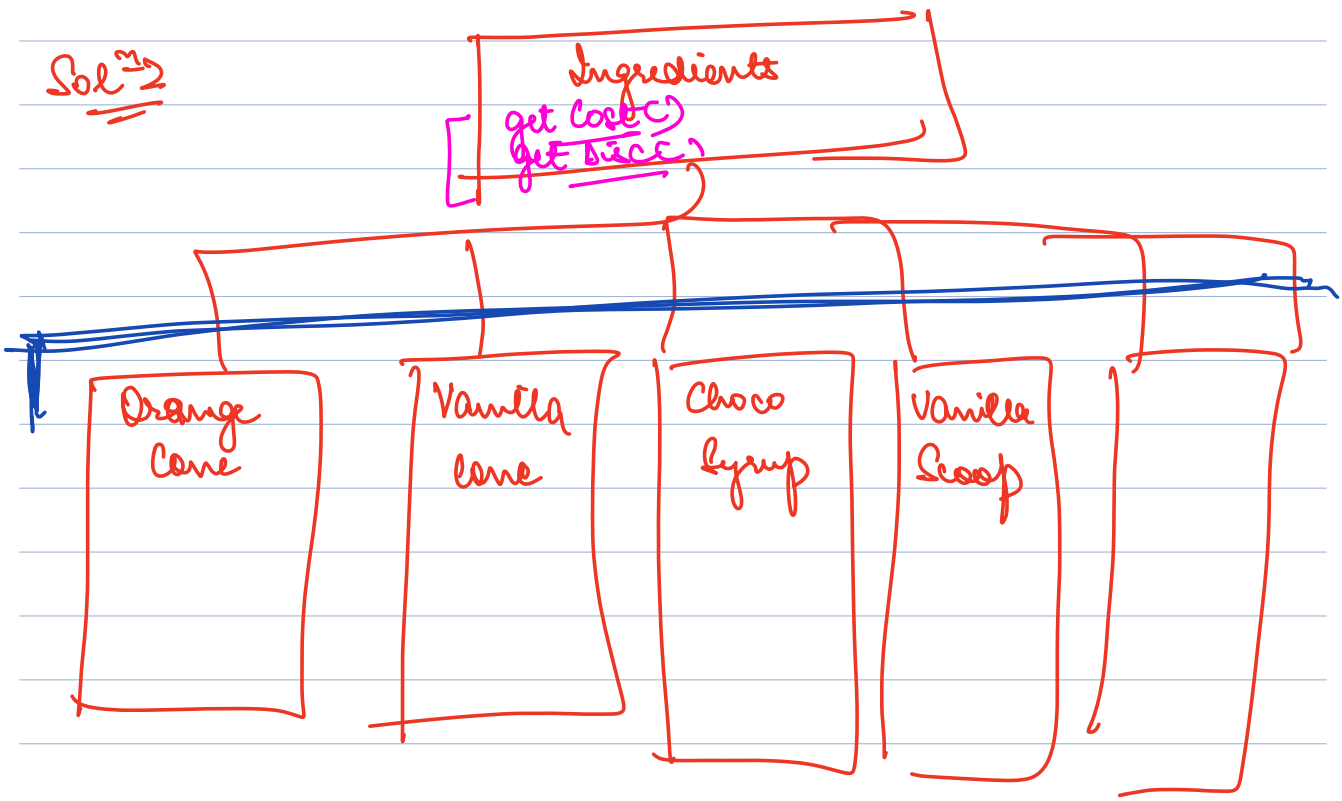

IceCream
list < Cone >
list < Topping >
getCost() {
// go through all cone + topping
}
getDisc() {
// go through all cone + topping.
}

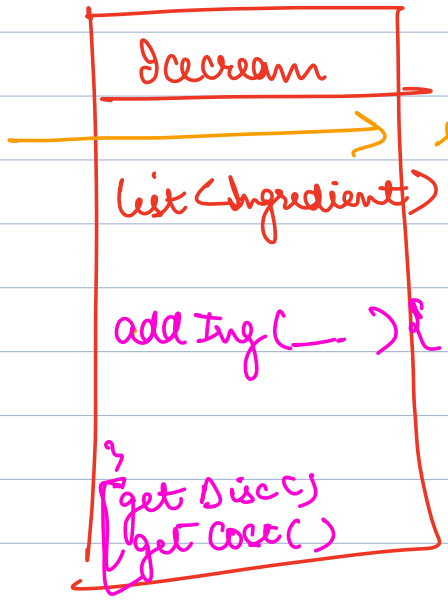





Cheese Burst

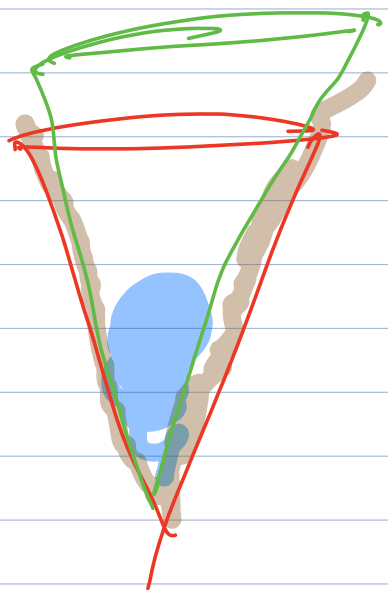
Sol =>


Ingredients
get Cost()
get Dice()
Orange Cone
Vanilla cone
Choco Syrup
Vanilla Scoop



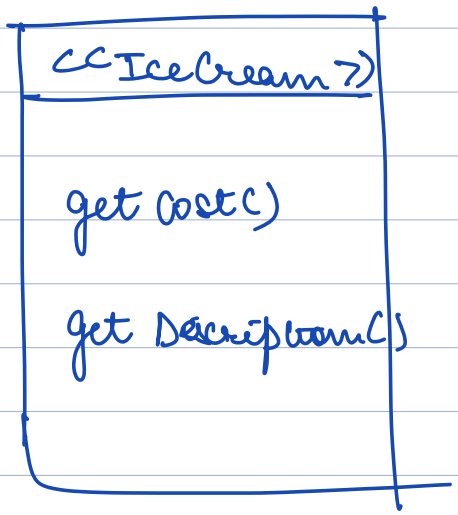
e-x : Keep putting things in the cone!

DECORATOR

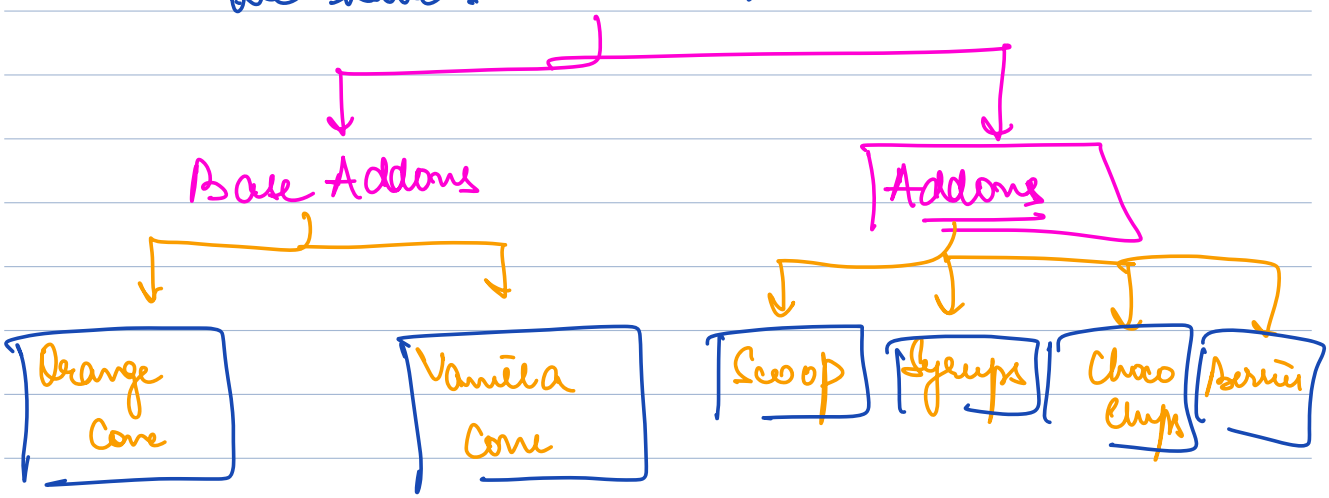


- Right from beginning I had Ice Cream Cone
- → whenever I added something new ⇒ cost, desc of icecream cone changed.

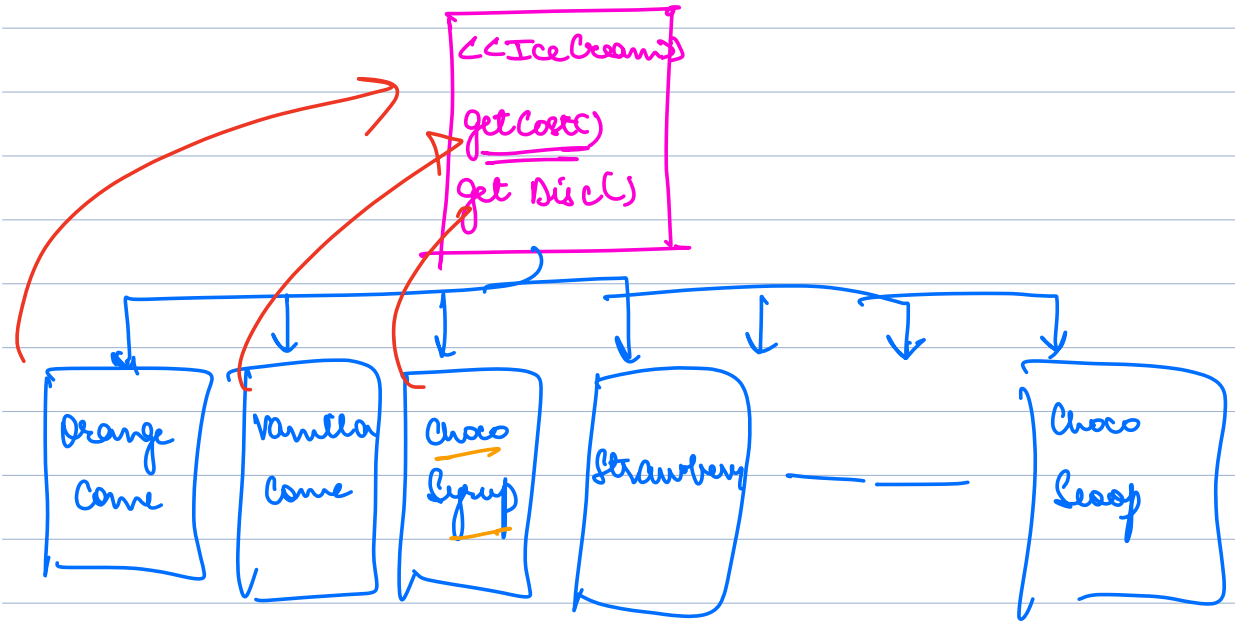
Step 1 : Define an interface / abstract class that represents the thing that we are constructing



Step 2 There are 2 kinds of ingredients that we have.



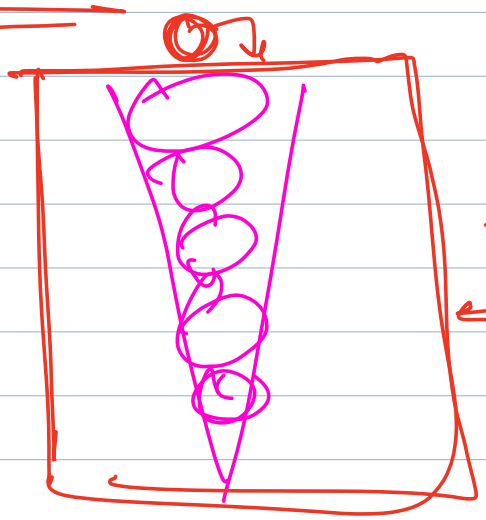
For each of these create a class that implement the interface



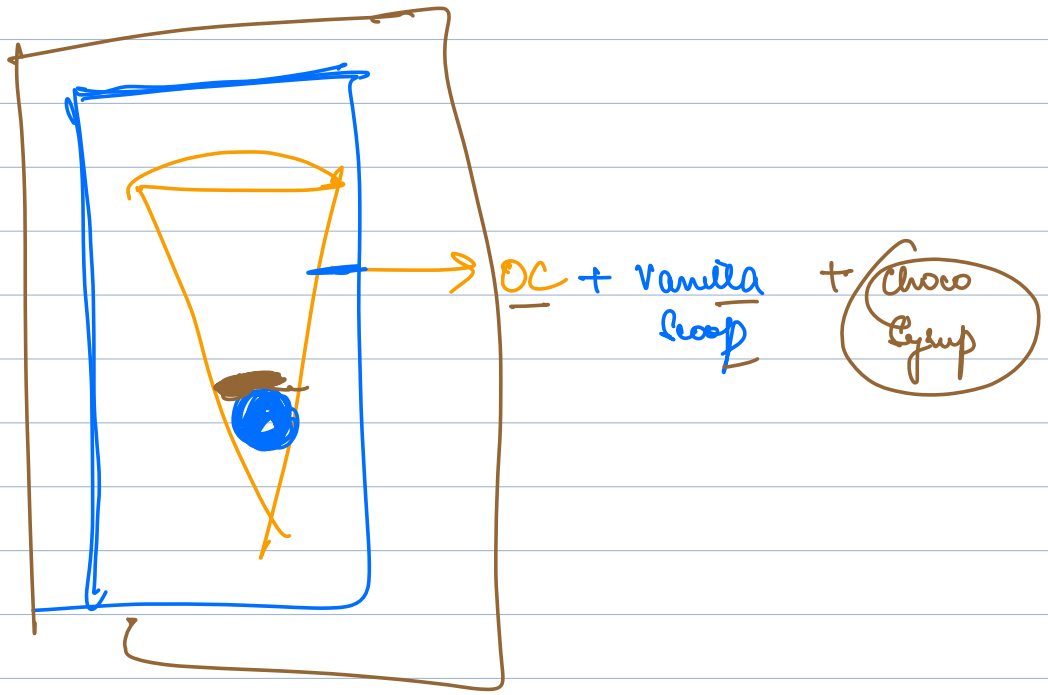
ONLY A BASE ENTITY

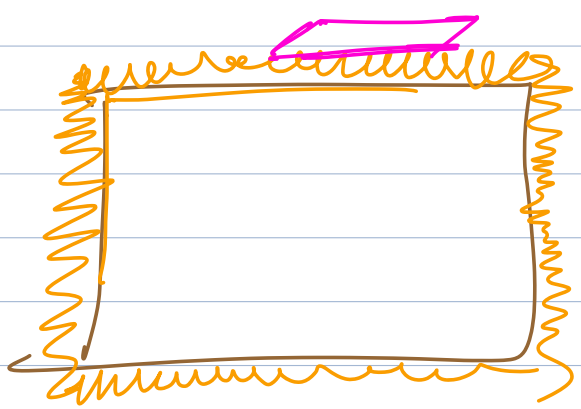
getCost() → cost of that entity
getDisc() → name of cone.

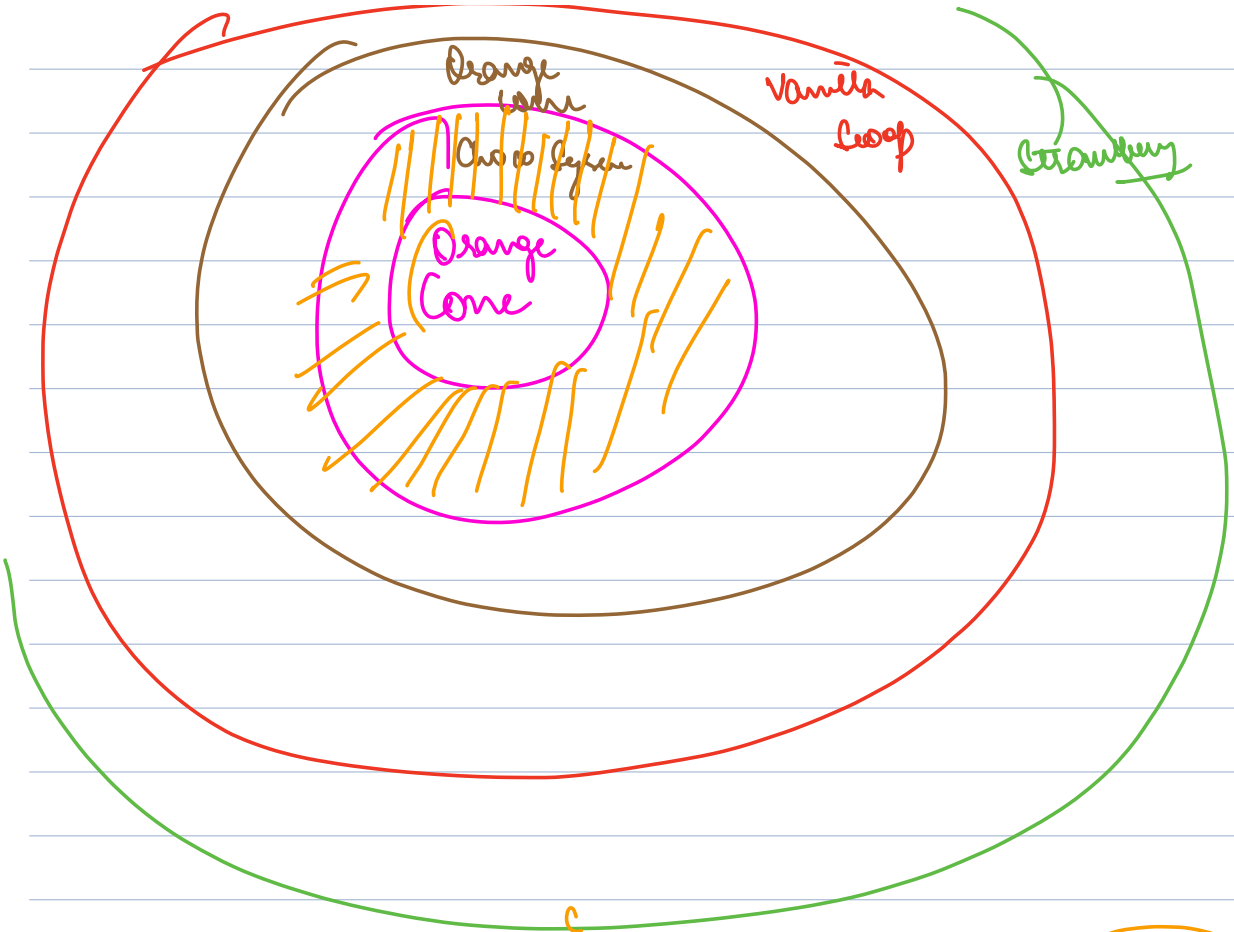
AFTER ADDON

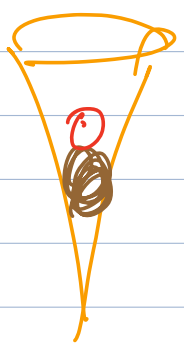


If we have a scenario where we add properties/features to a base entity at runtime where the final output depends on the output of base, consider using decorator design pattern









<< IceCream >>

Orange Cone

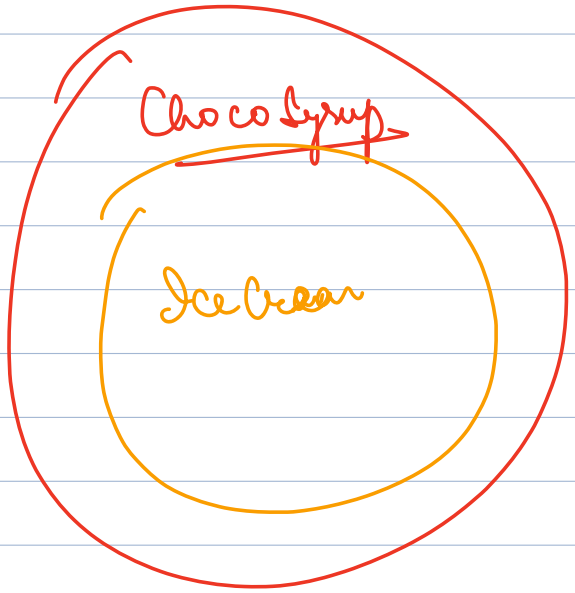
```
getCost() {
    return 10;
}

getDisc() {
    return "oc";
}
```

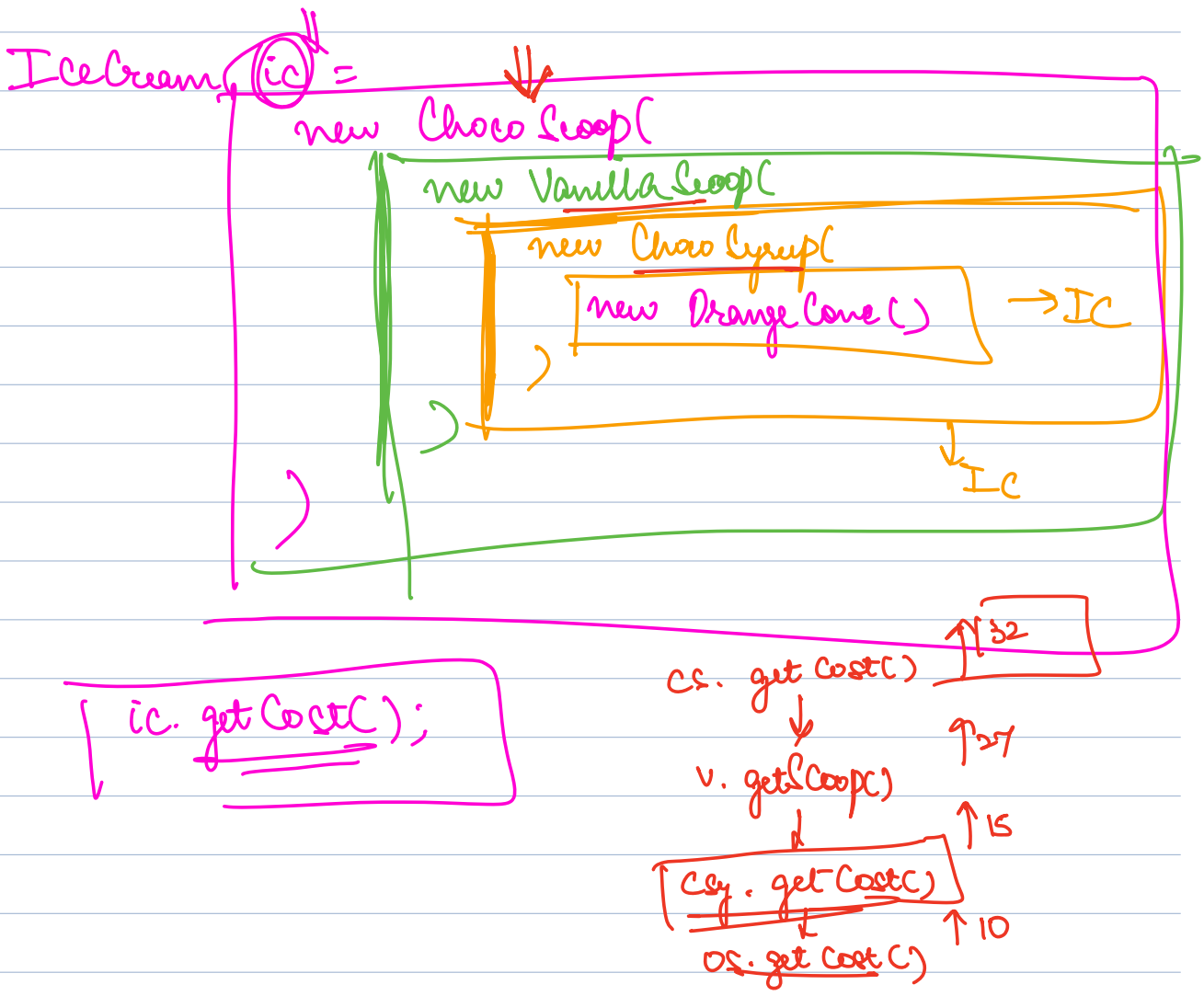
5

ChocoSyrup

→ IceCream

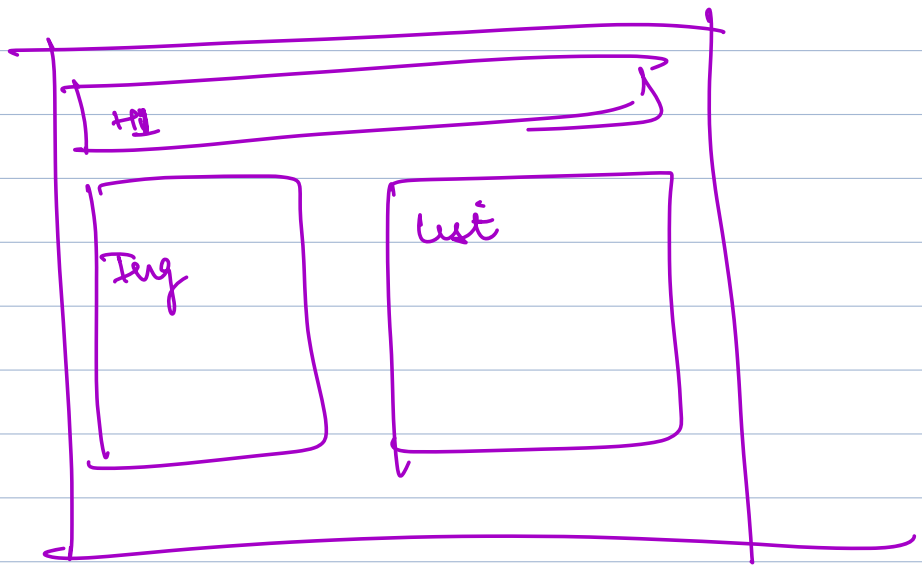
```
ChocoSyrup(ic) {
    this.ic = ic
}

getCost() => ic.cost + 5;

getDisc() => ic.disc + "choco syrup";
```

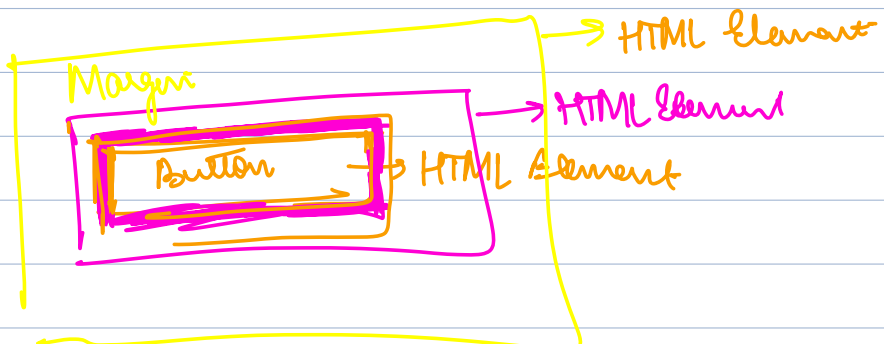

ChocoSyrup
IceCream


IceCream ic =
new ChocoSoop(
new VanillaSoop(
new ChocoSyrup(
new OrangeCone()
→IC
)
)
IC
)
ic.getCost();
cs.getCost()
32
v.getSoop()
27
cy.getCost()
15
os.getCost()
10

Decorator Design Pattern

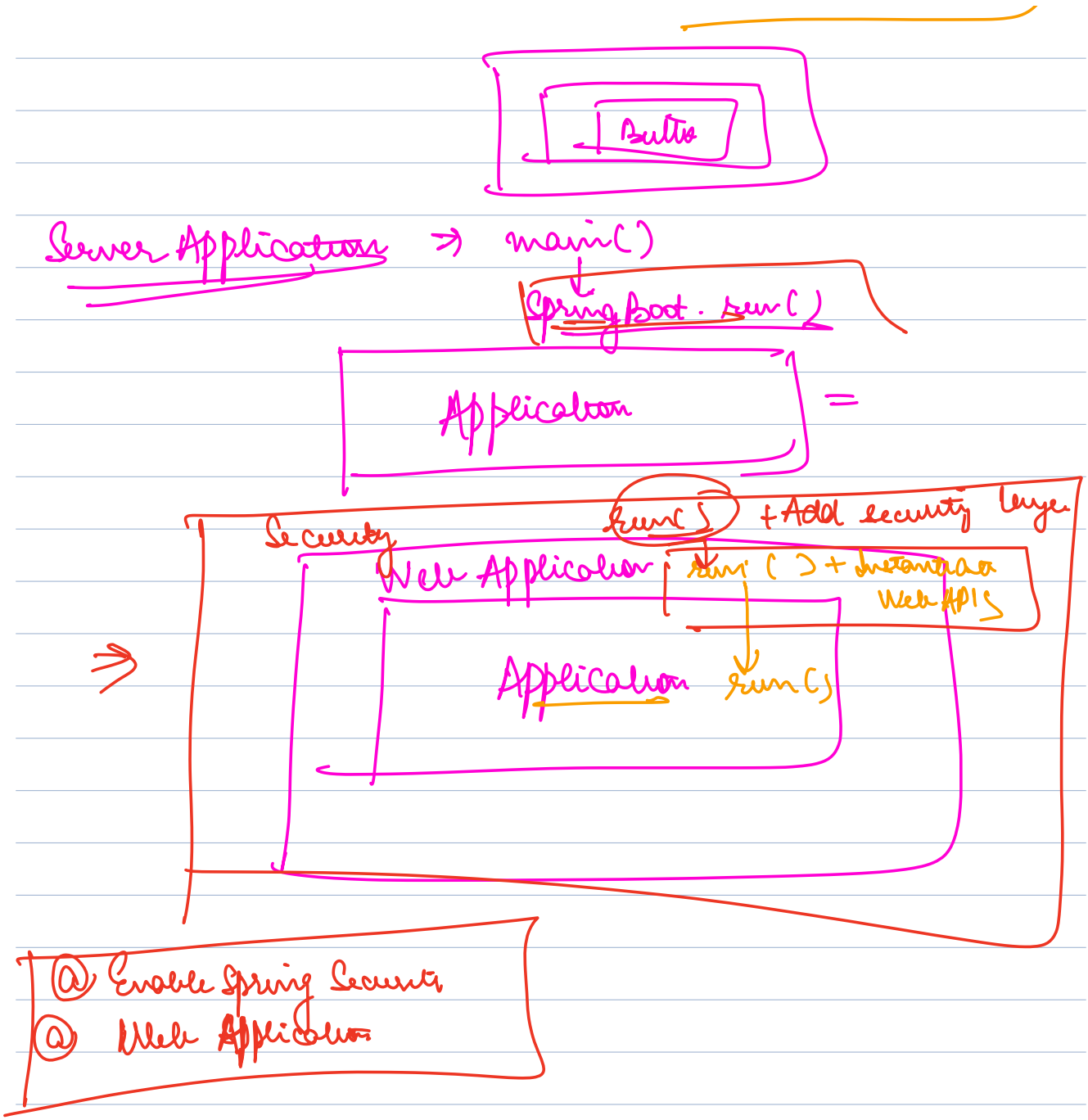
If we have an entity to which we may want to change behaviours / add to its behaviours at runtime, consider using decorator.

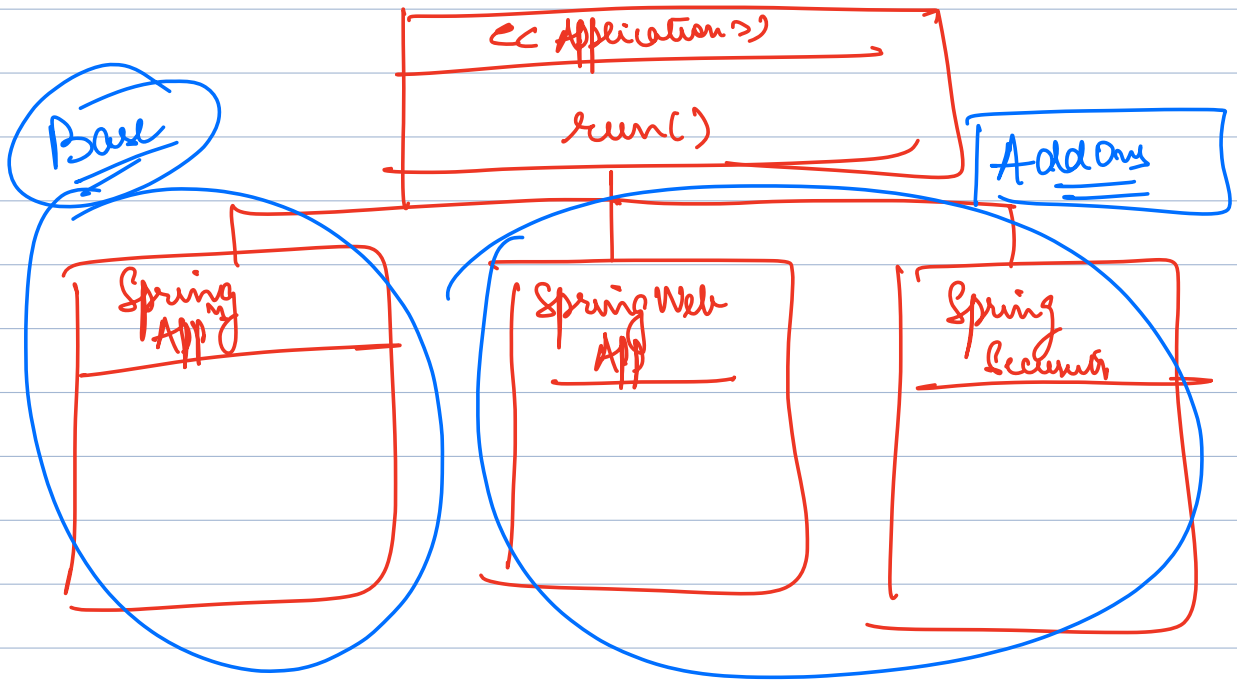





```
<<HTMLElement>>
display()
```

```
new Margin(
    new Border(
        new Button()
    )
);
```


Server Application
main()
SpringBoot.run()
Application
=
Security
run()
+Add security layer
Web Application
run() + Instance of Web APIs
Application run()
@ Enable Spring Security
@ Web Application
@ lru_cache()
— Decorator



If a class is both base + addon, it will need to have 2 cons:

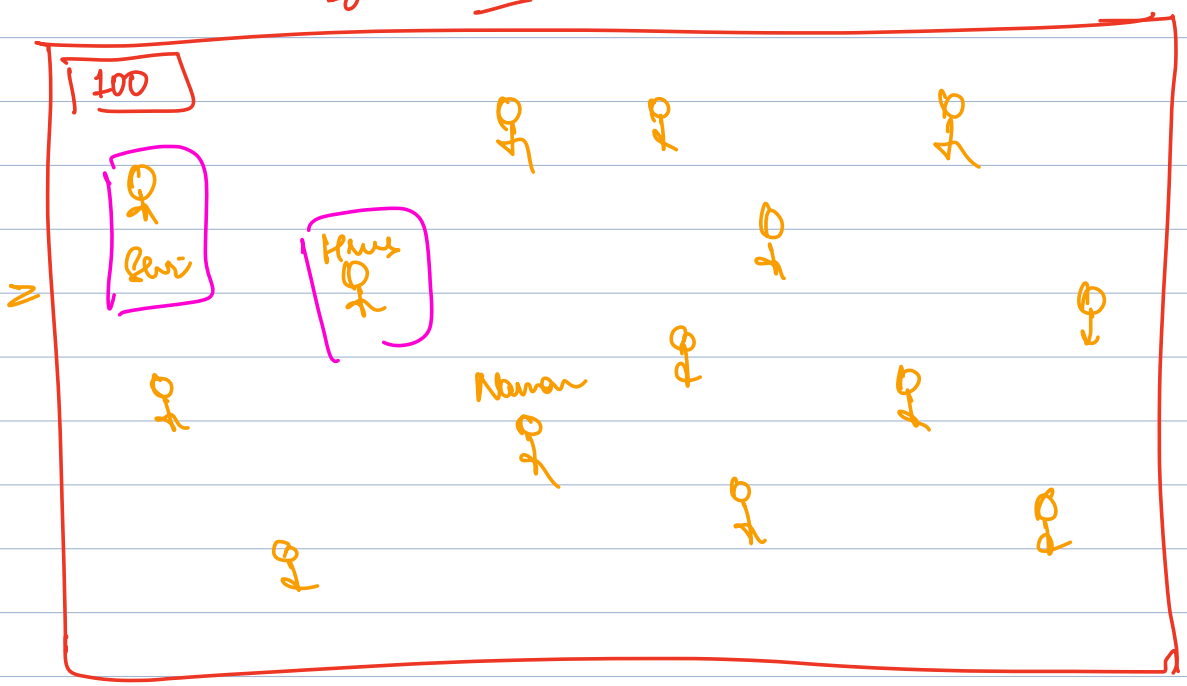
1. Empty constructor
2. constructor that takes param of interface.

A good codebase should avoid NPE

```
Orange Cone
-IceCream ic;
OrangeCone(){}
OrangeCone(IceCream ic){
    this.ic = ic;
}
getCost(){
    if (ic==null)
        return 10
    return 10 + ic.getCost()
}
```

Flyweight Design Pattern

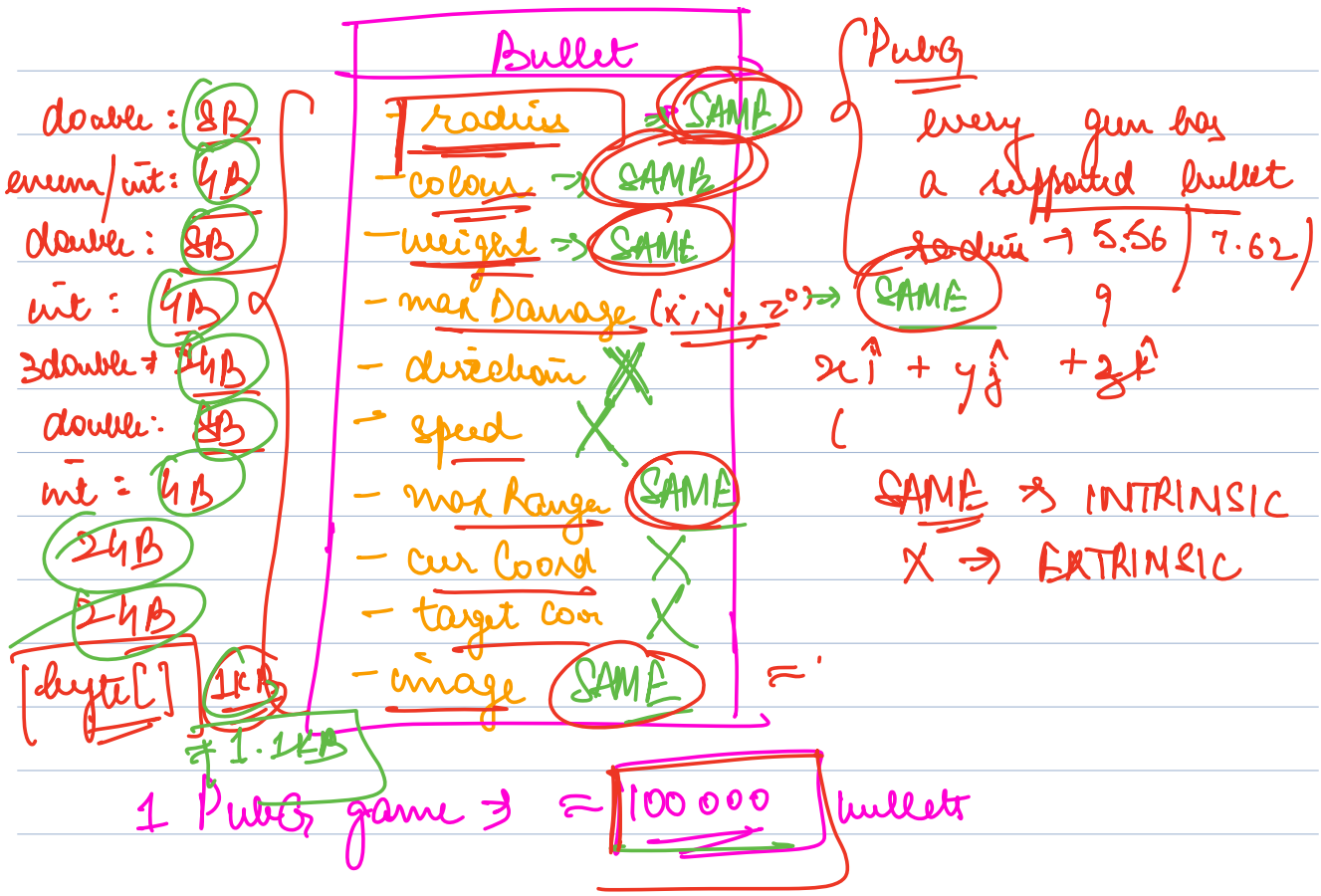
→ Engineer building UI of an online multiplayer Battle Royale game
Eg PubG



→ at the end only 1 player will be alive ⇒ WINNER

2 guns / player
300 bullets / player at a time

→ Complete State of the game is downloaded to the machine of every player
→ Changes of the game are transferred to every machine



Total RAM used by bullets

$\Rightarrow 1.1 \times 10^3 B \times 10^5$

$\Rightarrow 1.1 \times 10^8 B$

$\rightsquigarrow 0.11 GB$

$= \approx 100 MB$

OBSERVATION : Even though we have 100,000 bullets, not all bullets are completely distinct from each other.

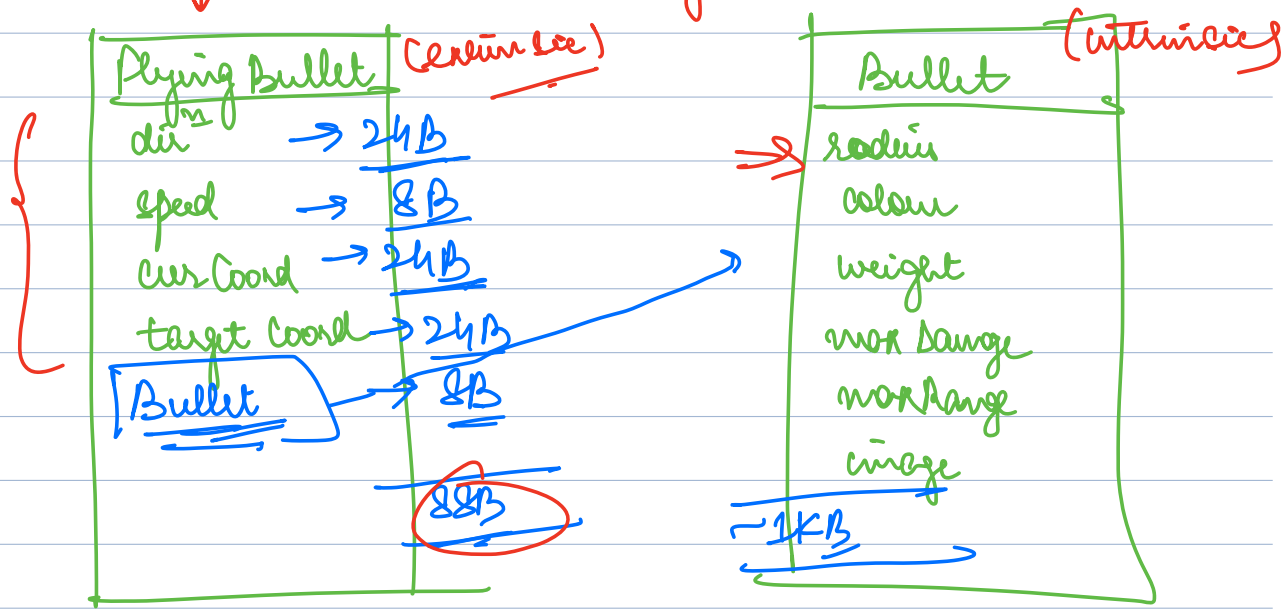
PUBG $\Rightarrow$ 10 diff types of bullets

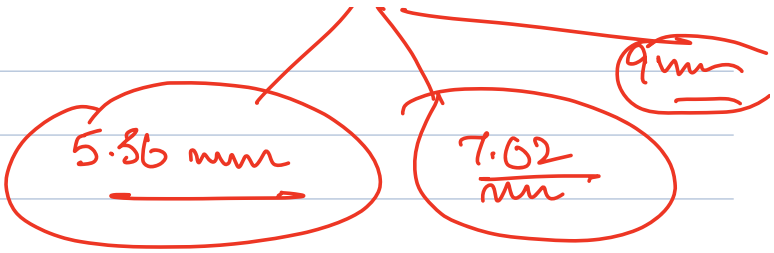
Often we have classes that have 2 types of properties:

- Extrinsic (Values of which might change with time)
- Intrinsic (The value of which remain same across all obj)

Flyweight: If you have a class whose objects demonstrate intrinsic as well as extrinsic properties AND if you notice extra memory usage because of that, consider using flyweight

(1) Divide that class into 2 classes
   a) with only intrinsic
   b) with only extrinsic





PubG has 10 types of bullets
⇒ 10 Bullet obj
⇒ 10×1 KB ⇒ 10KB
There are total 100000 Bullets
⇒ 100000 BB obj
⇒ 100000×8B
⇒ $8 \cdot 8 \times 10^6$ B ⇒ 8.8MB

Total Space ⇒ 8.8MB + 10 KB ≈ 9MB