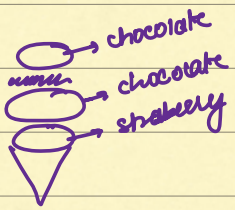
AGENDA

✓ 1.> Decorator design Pattern (Coffee machine)

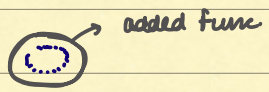
✓ 2.> Flyweight design Pattern (Publi)

structural dp.

start by 9:05 PM IST



Ice cream



*] DECORATOR DESIGN PATTERN:

definition: Provides wrapper to existing class which allows to have added functionality.

eg: working at Baskin Robbins as SWE

→ "Build ice-cream ordering system"

a.> user can select any type of cone
- chocolate
- waffle
- Plane cone

b.> user can add flavours of ice-cream one by one
vanilla / chocolate / cherry etc.

→ basically: user can order customised ice-creams.

Q.> You need to build such Application which works on similar concept.

Requirements:
1. Build Ice cream
2. Get final cost of ice cream
3. Print list of ingredients / description

terms:

→ cone / topping / ice-cream

SOLUTION:

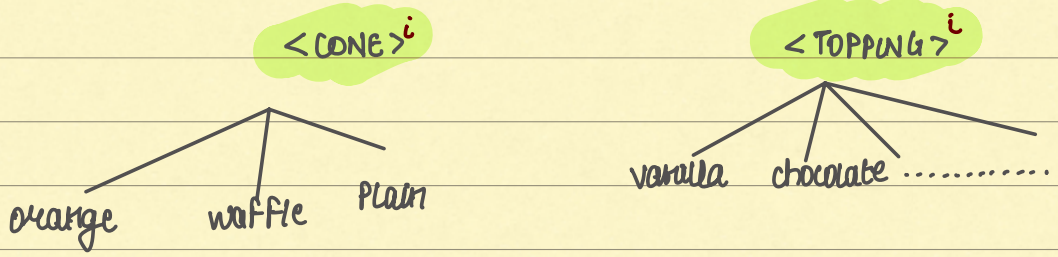
1.) CONE

2.) topping

icecream

- List<topping>
- List<cone>

cone / topping can be an Interface ✓



Icecream is basically

```
class Icecream {
    List<cone>
    List<topping>
}
```

cone/Topping is basically:

```
cone/topping {
    getCost() ✓
    getDescription() ✓
}
```

- -> Ice-cream can have many toppings
- -> Ice-cream can have many cones (assume)

getCost(){

USECASES:

1.) Get cost of ice-cream- }

cost (cone) + cost (topping)

// Method:

int getCost() {

// add cost of all →

and return

}

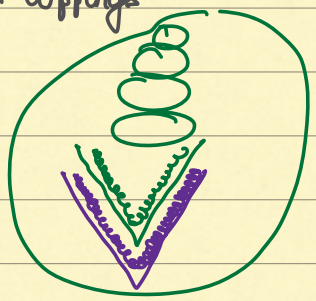
2.) Get description

Go through each cone / topping & Get data.

string getDescription() {

// Go through all cone + toppings

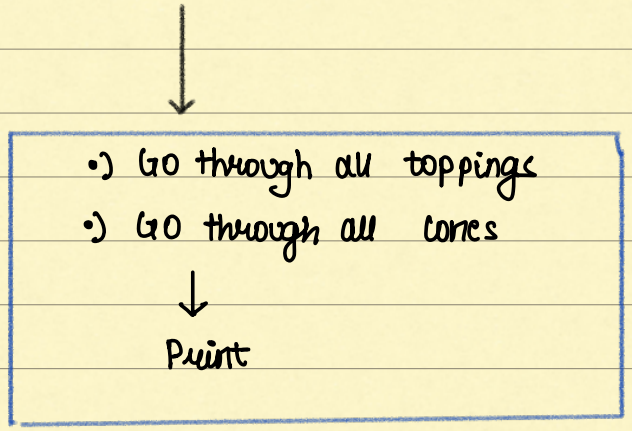
}



Q.) ANY EDGE CASE ???

```
Icecream i = new Icecream();
i.getDescription();
```

Flow:



VERY WIERD USECASE:

→ can Anyone order like this :

Now: calling `i.getDescription()` : would it give correct output ???

All toppings → All cones

Another Good Example: Pizza

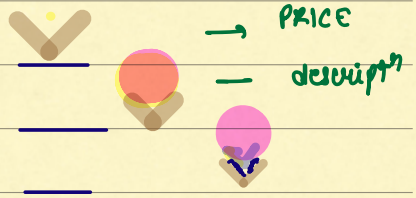
① Normal Pizza
② Pan Pizza
③ Cheese burst Pizza

∴ Above SOLUTION WON'T WORK...

*] BETTER SOLUTION:

cases:

→ ① single cone
② single cone with scoop
③ Multicone with Multiscoop



still ice-cream.

→ Just Price and desc̲ription change
as because : things added on top.

*] STEPS:

S1.) Define Interface / Abs class that represents things that we construct

Here: ice-cream

<Ice-cream>
- getCost()
- getDescription()

① Base (one)
② Add-ON

② kinds of ingredients MAJORLY:

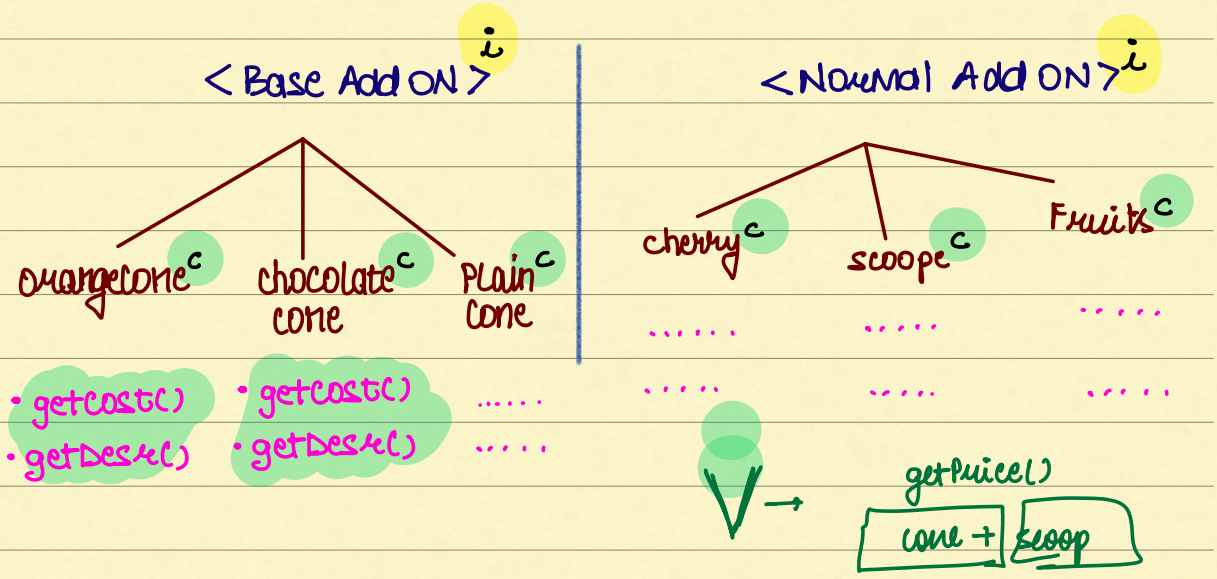
→ Observation :

✗ Base ENTITY
✓ ADD-ON

have ② Ingredients

1.> base add ons — different type of cone
2.> Normal add ons — exist over base add-ons.

S2) for Each of Ingredient: create Interfaces > specific classes.



BaseEntity:

- getCost() — returns cost of cone
- getDesc() — returns desc of cone

Normal Add ON:

:) already had an ice-cream ⊕
Add Extra anything

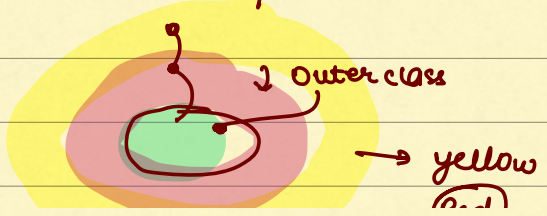
- getcost() - existing + New add-ON.
- getdesrip() -

Decoratore dp: case where we add properties to Base Entity at Runtime → Final output depends on base.

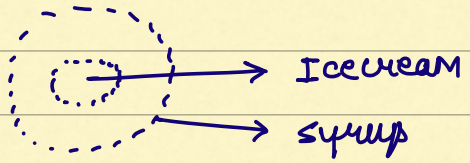
Base
↓
orangecone
getcost()
getDescp()

Add ON:
X exist by self
:) add this on base object
cone + chocolate scoop.

Any Add ON will always be wrapped ON Ice-cream class ONLY....



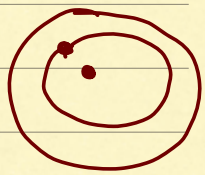
How TO WRAP classes:



i= new chocoscoop
i= new orange()

```
class chocolatesyrup{
    Icecream i;

    chocolatesyrup(Icecream input){
        this.i = input
    }
}
```



```
chocolatesyrup cs=
        new chocolatesyrup(i);
```

Method Implementations:

```
getCost(){

return i.getCost()+
        this.cost
}
```

```
getDescription(){

return i.getDesp() +
        this.desc;
}
```

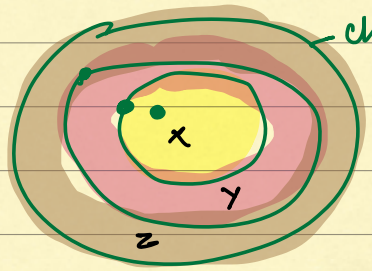
Note: Handle Null cases

★] ADVANTAGE:

creating Ice-cream object:

- start with base Entity
- Keep wrapping Addons.

Icecream ic =



x = 2
y = 4
z = 6

x.getcost() = 2
y.getcost()
z.cost() z.cost + y.getcost()
6 +
(6)
= (12)

Y y = new y(x);

getcost() {

return x.getcost() + this.cost();

}

Example:

① html button

plain button / border button

② copy assignments in college...

steps:

① create base class

② create Other classes

① base class. Implement methods

② Normal Addons - take param in constructor

share code link

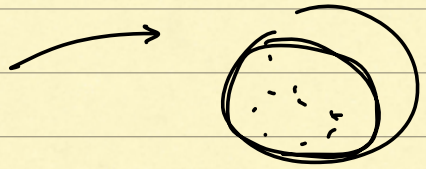
CONTRA

*] FLYWEIGHT DESIGN PATTERN:

definition : Re-uses already existing similar Objects by storing them

eg: build UI for online Multiplayer Game. (PUBG)

Pubg → battle Royal Game



HOW ONLINE GAMES WORK:

1.) Complete state is downloaded to Machine/phone of every Player

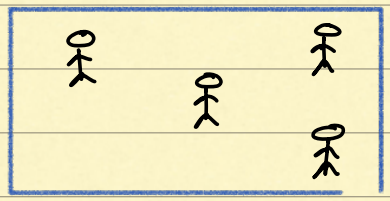
↳ start Game takes time to load.

2.) when Game starts, changes of Game are transferred to all SERVERS (all players)

i.e. bullets/ Guns/ cars (Info) about Game is Maintained ON UI.

Bullet

*] CLASSES:



- PlayGround ✓
- Player ✓
- Gun ✓
- BULLET * ✓

(A)

Attributes of bullet:

- color - STRING 6char = 6×1 = 6bytes
- Range → double → 8 bytes (Max)
- size → double 8bytes
- velocity → double 8
- type → SMALL INT 1 byte
- weight → double 8byte
- damage → INT 4
- target-coordinate → (Xi, Yi, Zi) 24
- bodytype → SMALL INT 1

68 bytes

8+4+24 = 36

1+24 = 36

36 bytes

(3)

avg size of 1 bullet: 68

How many bullets in (1) Game are required ON AVG:

① No. of.... players 100 Players

avg bullet count = 1000

⊕

② we need info on bullets lying on field....

Assume total: 100,000 bullets.

Now; total RAM used: 100,000 × 64 bytes = ~ 100 Mb

4 Mb

stored in RAM

Not good Idea.....

*] BULLETS:

(A)

Types of bullets - many

A/B/C/D.....

→ consider any (1) type of bullet:

same attributes	different
Radius	direction
name	speed
color	origin
weight	destruction

In such cases we can use flyweight design Pattern.

Whenever we have classes that have (2) types of Properties

1.) Intrinsic
2.) Extrinsic

Intrinsic	Extrinsic

value remains same for all objects | diff value for diff objects

Hence, if we have such usecase where object takes unwanted extra space

↓

consider using flyweight design pattern.

*] STEPS:

1.> divide class into (2) parts
   a) with only Intrinsic — remain same
   b) with only Extrinsic — dynamic

E

FlyingBullet

direction	24 bytes
speed	8 bytes
origin	24 bytes
destination	24 bytes
	80 bytes

I — same

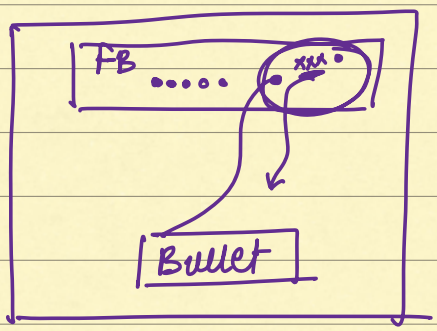
Bullet

Radius	4 bytes
Name	8 bytes
color	4 bytes
	16 bytes

*] ADVANTAGE:

Gameclass → should have both Intrinsic / Extrinsic

```
class flyingBullet {
    Bullet b;
    .....
    .....
}
```



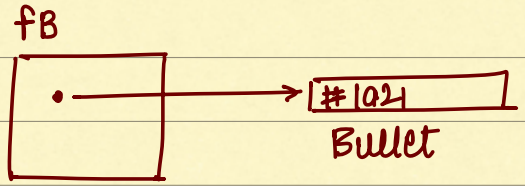
NOW; size of bullet:

(size of extrinsic attributes) + size of Bullet (Intrinsic)

80 bytes + 8 bytes

Bullet → stored as object (reference)

complete object ⊕ values is NOT STORED



bulletsize = ~ 1kb

FlyingBullet = 80 bytes

SIZE CALCULATION:

10 types of bullet used in PubG

10 Objects × $\frac{\text{bullet size}}{1\text{kb}}$ = 10kb

FlyingBullets = 100,000 × FlyingBullet weight

= 100,000 × 88 bytes = 8.8 Mb

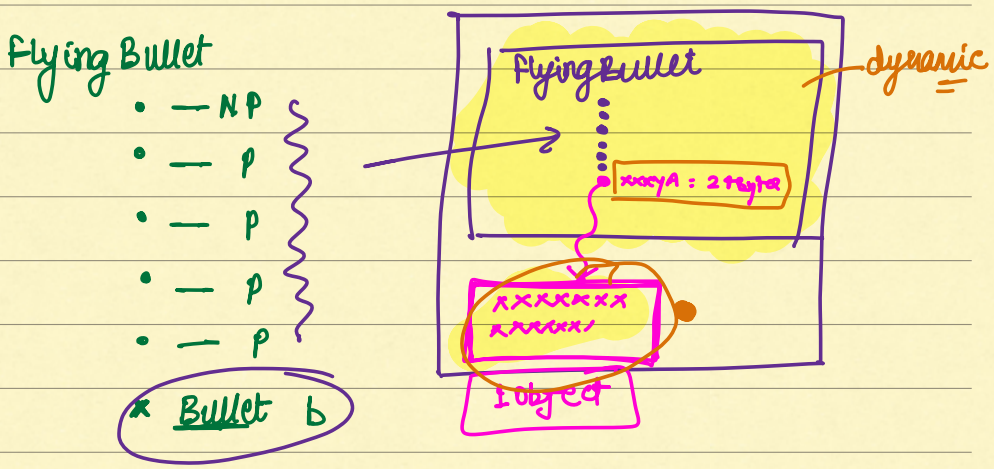
~9 MB

Earlier = 100Mb

Now = 9Mb

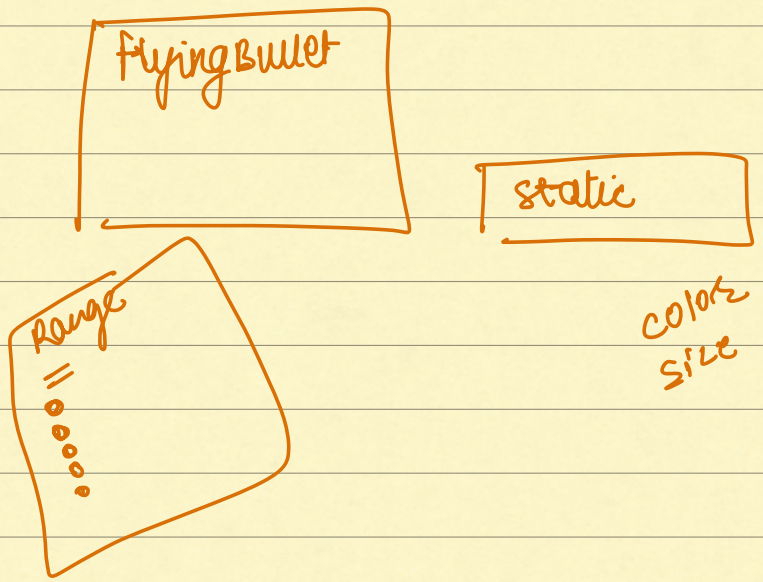
Note: Bullet class can be stored in Registry

Flyweight + Registry


FlyingBullet
NP
P
P
P
P
Bullet b
FlyingBullet
dynamic
xxxyA : 2type
xxxxxxxx
xxxxxxx
1 object

1 Bull

bull


FlyingBullet
static
color
size
Range