Agenda

How to approach design problem

1. Class diagram of Pen
2. Composition, Association, Aggregation

UML Diag

How to do class diagram

1. Visualization → Today's TPT, BMS, Parking lot
2. Nouns in Requirements → Splitwise.

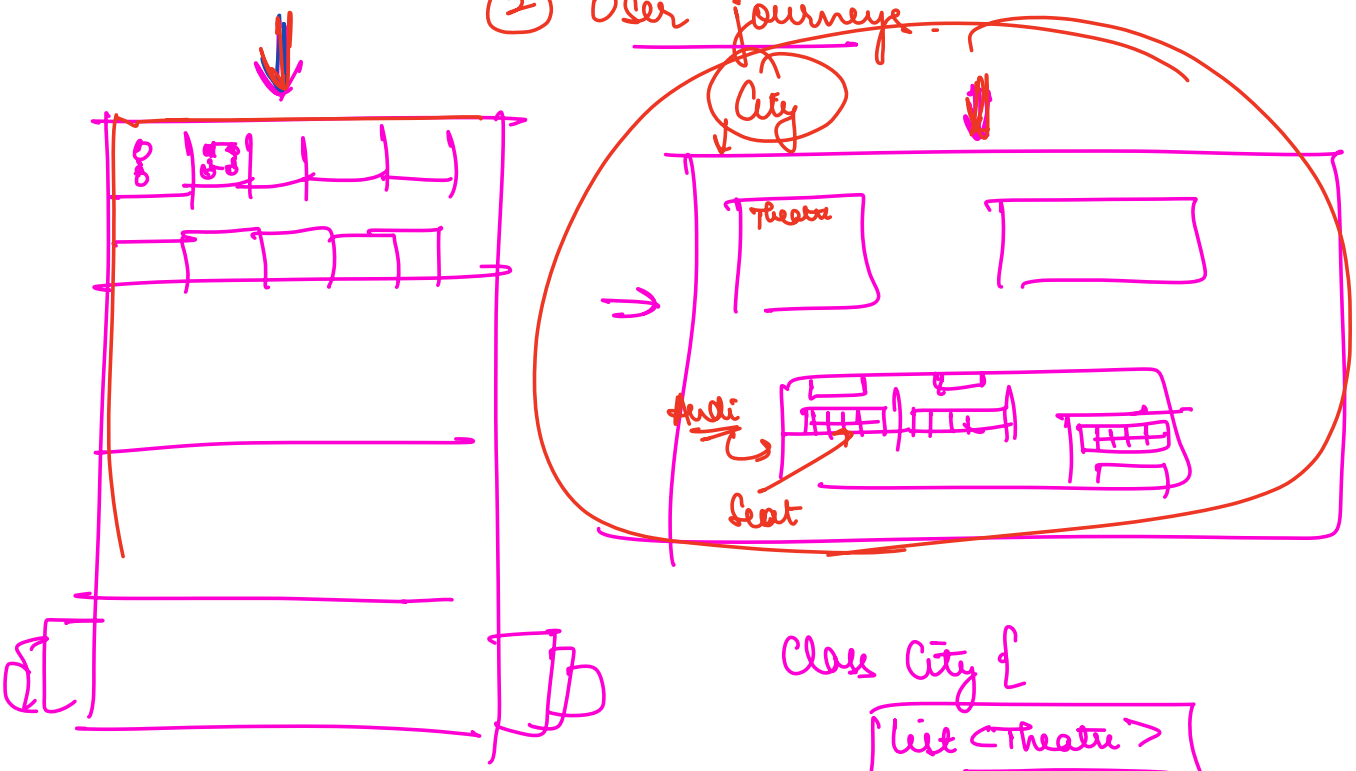
Expectations from a class diagram

1. Only expected to show classes for entities in the systems
   - → no need to mention controllers, services, repositories etc
   - things about which you will store info.
2. Any use of design pattern that you see in your appn

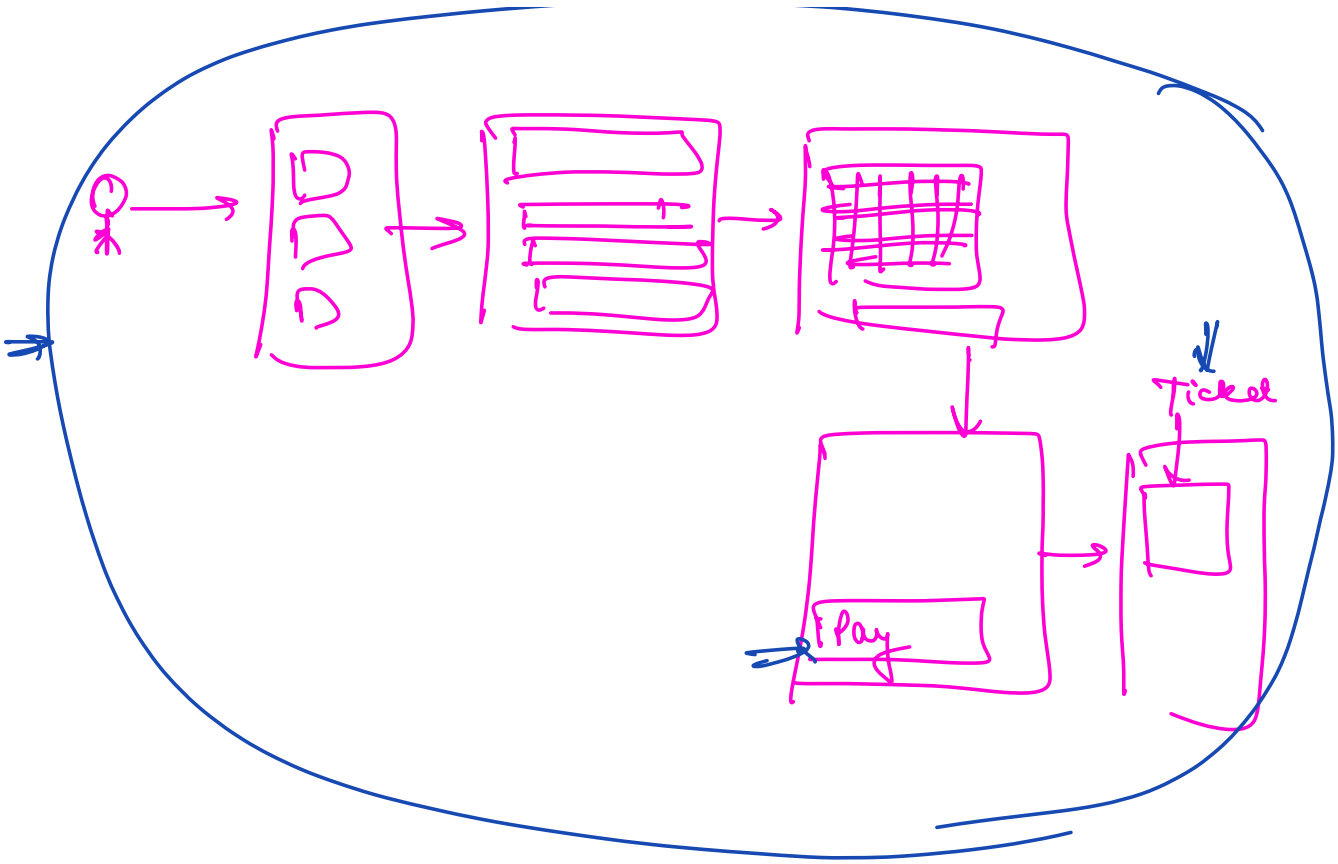
FIND CLASSES VIA VISUALIZATION

Visualize 2 things → Create a rough sketch

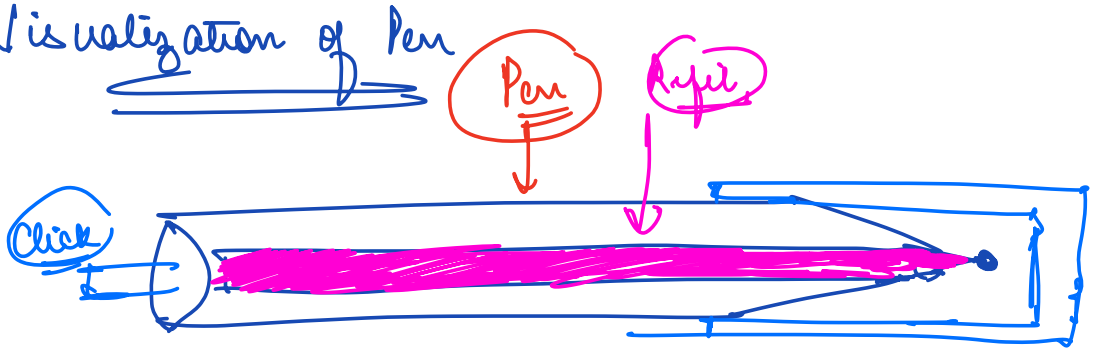
1. Physical structure / hierarchy of diff entities involved.
2. User journeys.

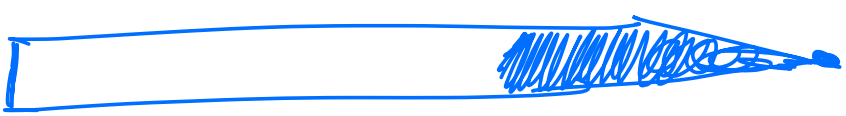



```
Class City {
    List<Theatre>

}
Class Theatre {
    private List<Audi>

}
```

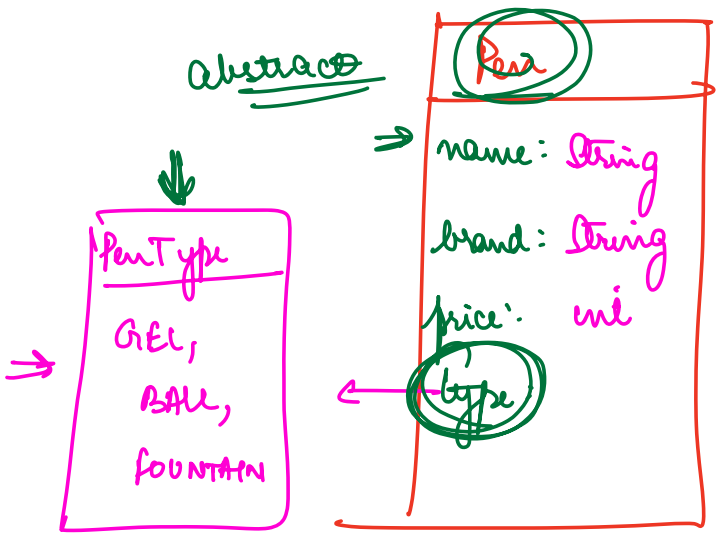


Visualization of Pen





→ Once visualized start writing classes from outside to inside.



→ Often an entity for which there are multiple types is defined abstract with a type attribute

if pen.type == ___

elif pen.type == ___

Factory

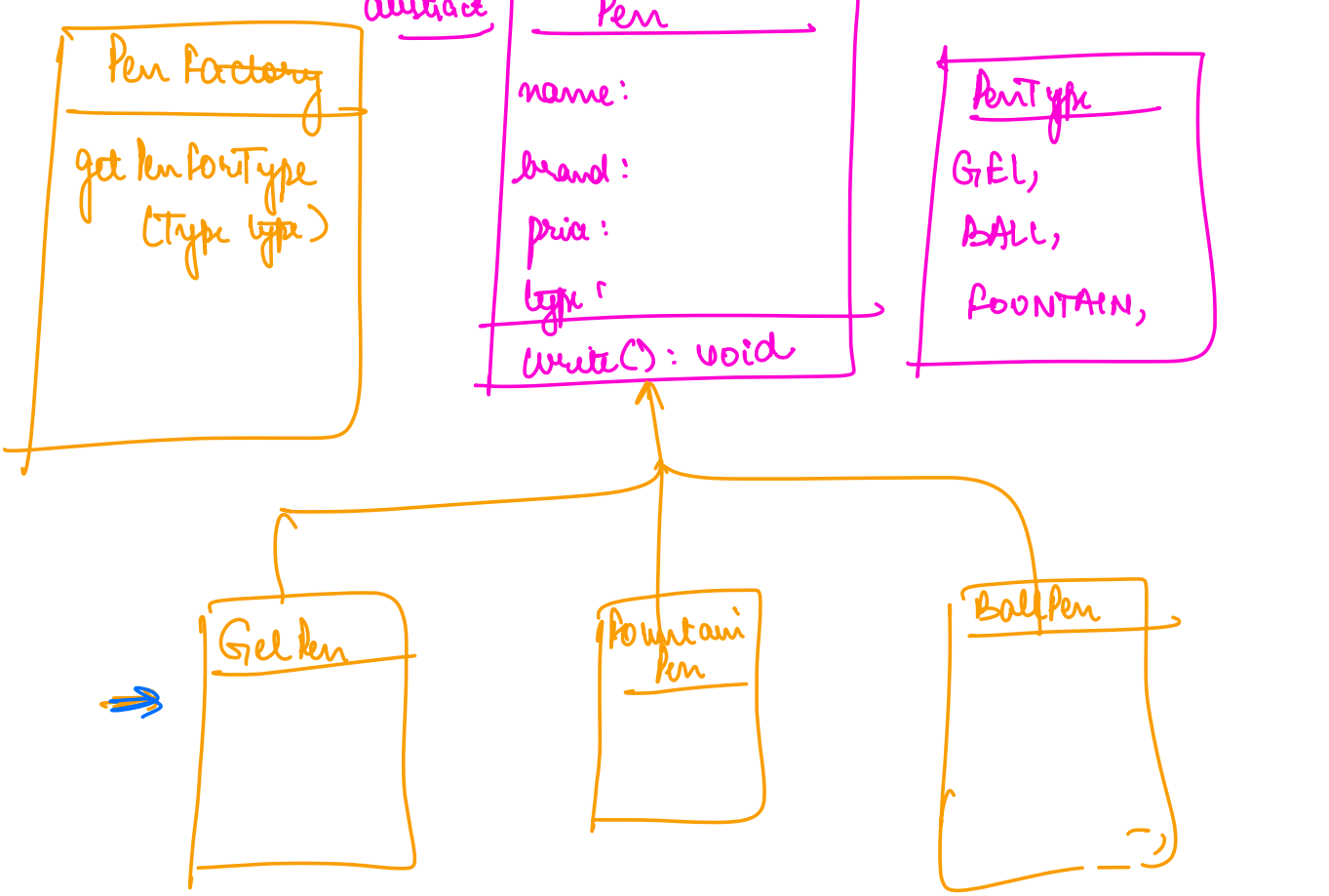
create Pen Of Type (GEL)

Whenever you have an entity for which there are multiple types, you often define that entity as abstract with a type attr.

① a bulk create factory

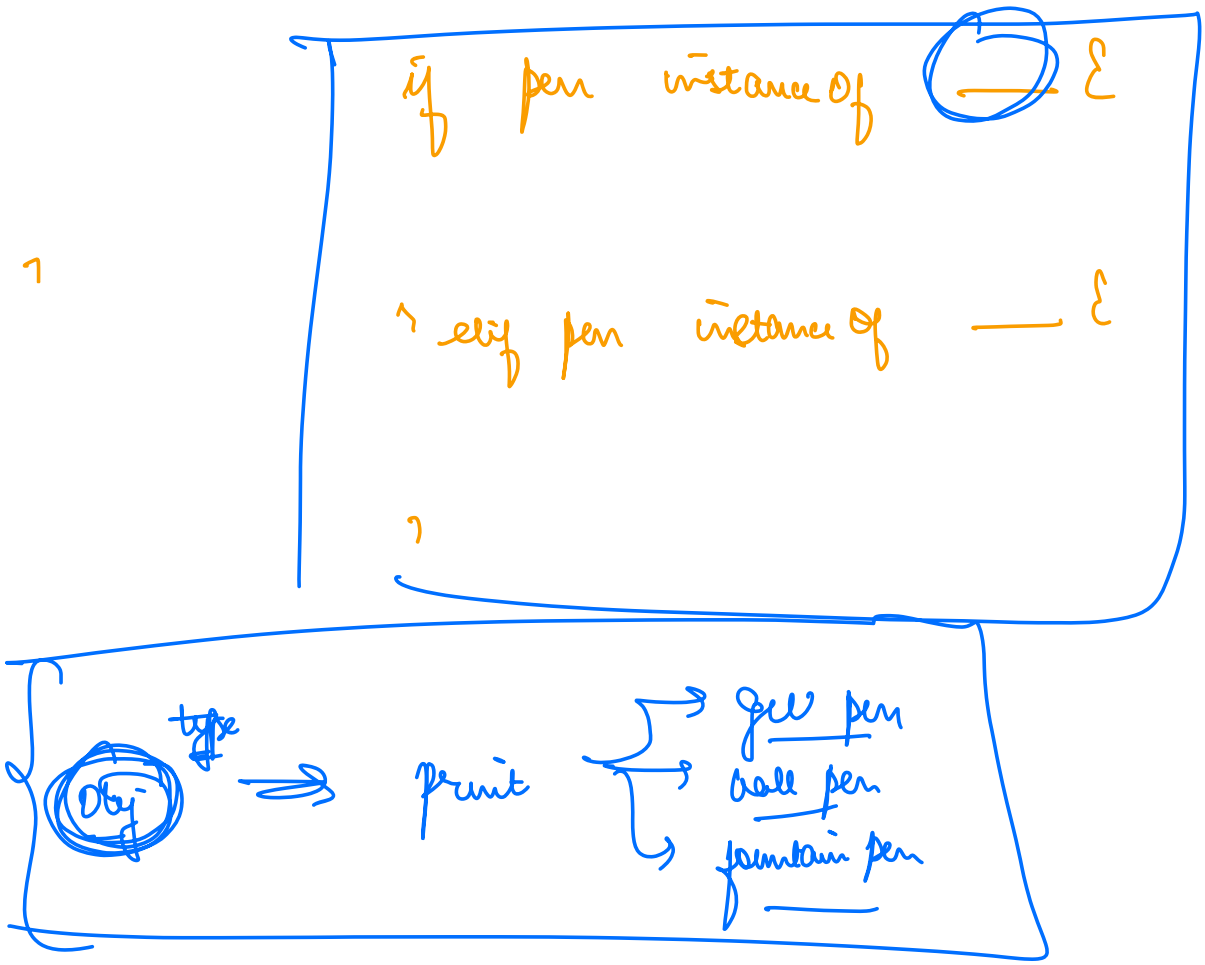
PenFactory {

createPenForType(PenType) {



Factory {

createPen for ( )



Switch (penTypes)

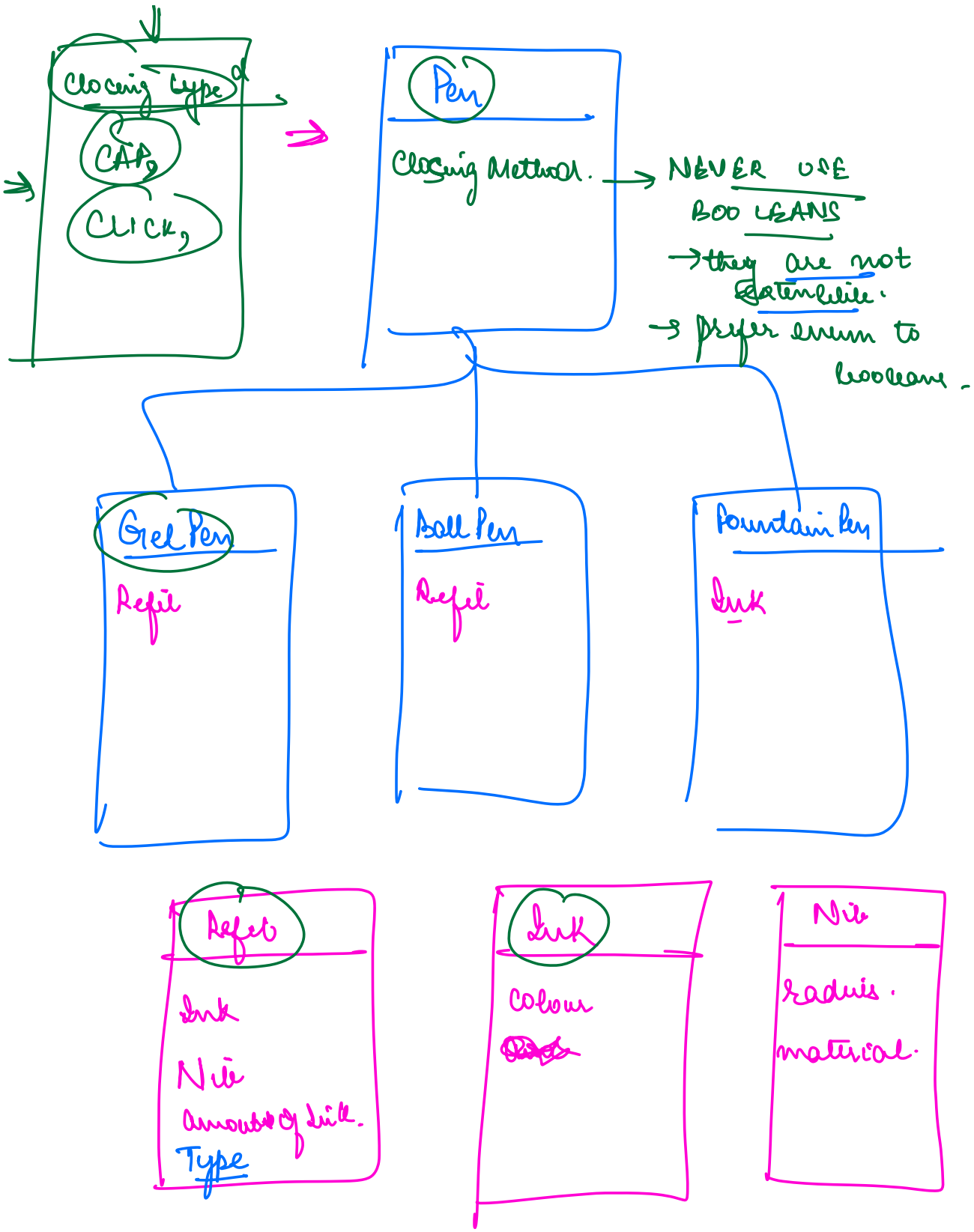
case GEL:

——

case BALL:

——

case


Closing type
CAP
CLICK
Pen
Closing Method.
NEVER USE BOOLEANS
→ they are not extensible.
→ prefer enum to boolean.
GelPen
Refill
BallPen
Refill
FountainPen
Ink
Refill
Ink
Nib
amount of Ink.
Type
Ink
Colour
Nib
radius.
material.

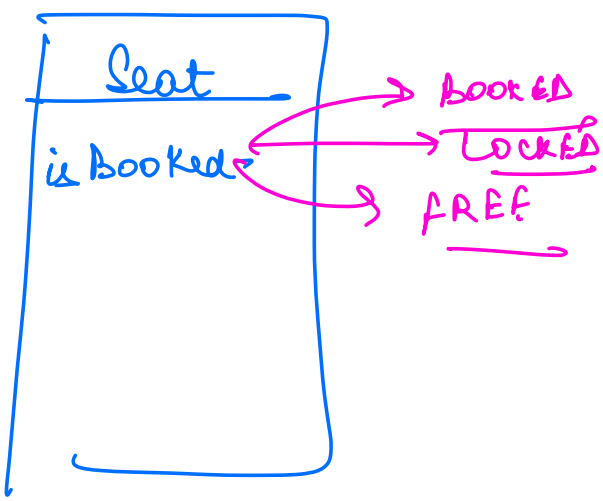
Uber Eats

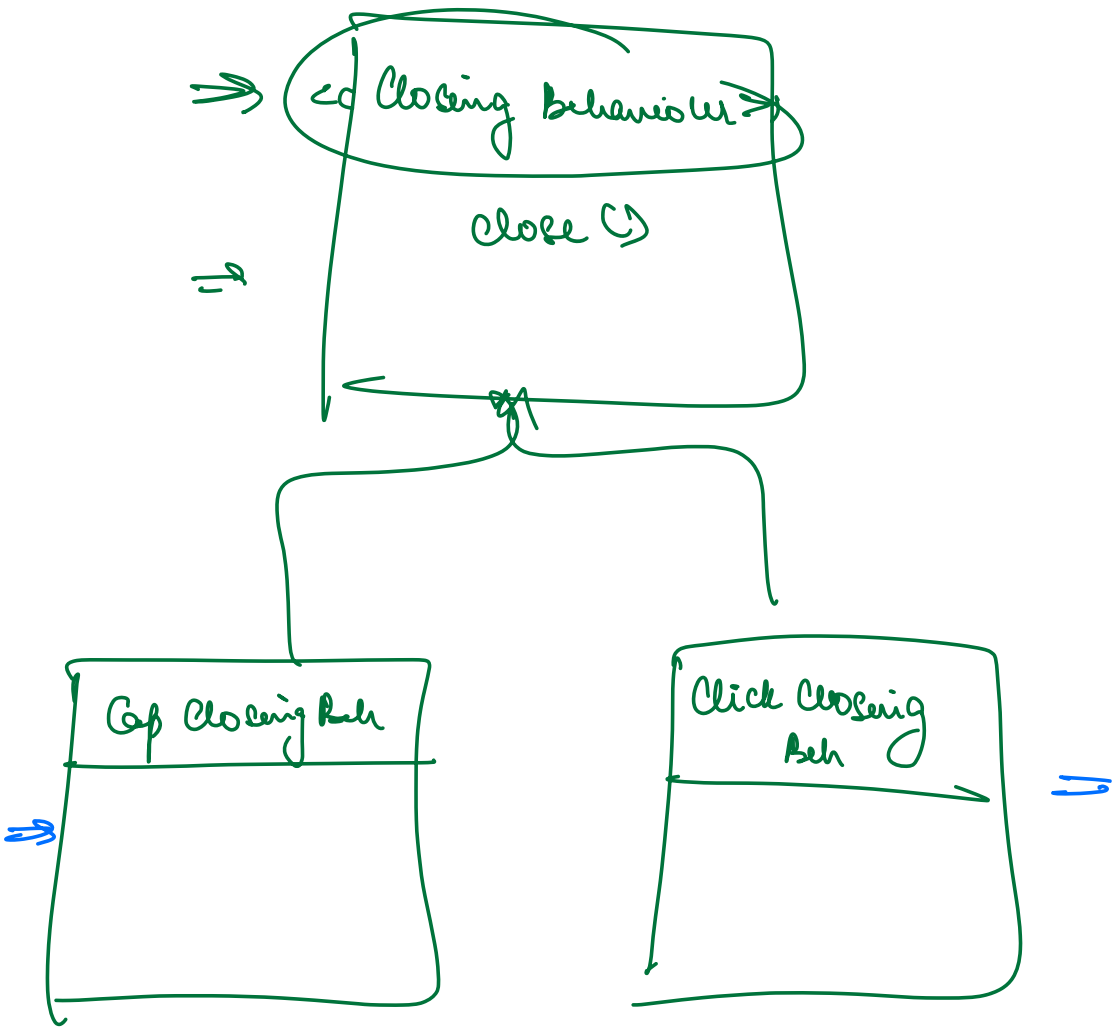
→ used PayTM for its payment gtw.

```
pay() {
    r = //response from PayTM
    ⇒ success = true.
    if r.status == FAILURE
        ⇒ success = false
```

PayTM

```
response {
    id: ____
    status: SUCCESS / Failure / PENDING
    time: ____
    amount:
```





PenFactory {

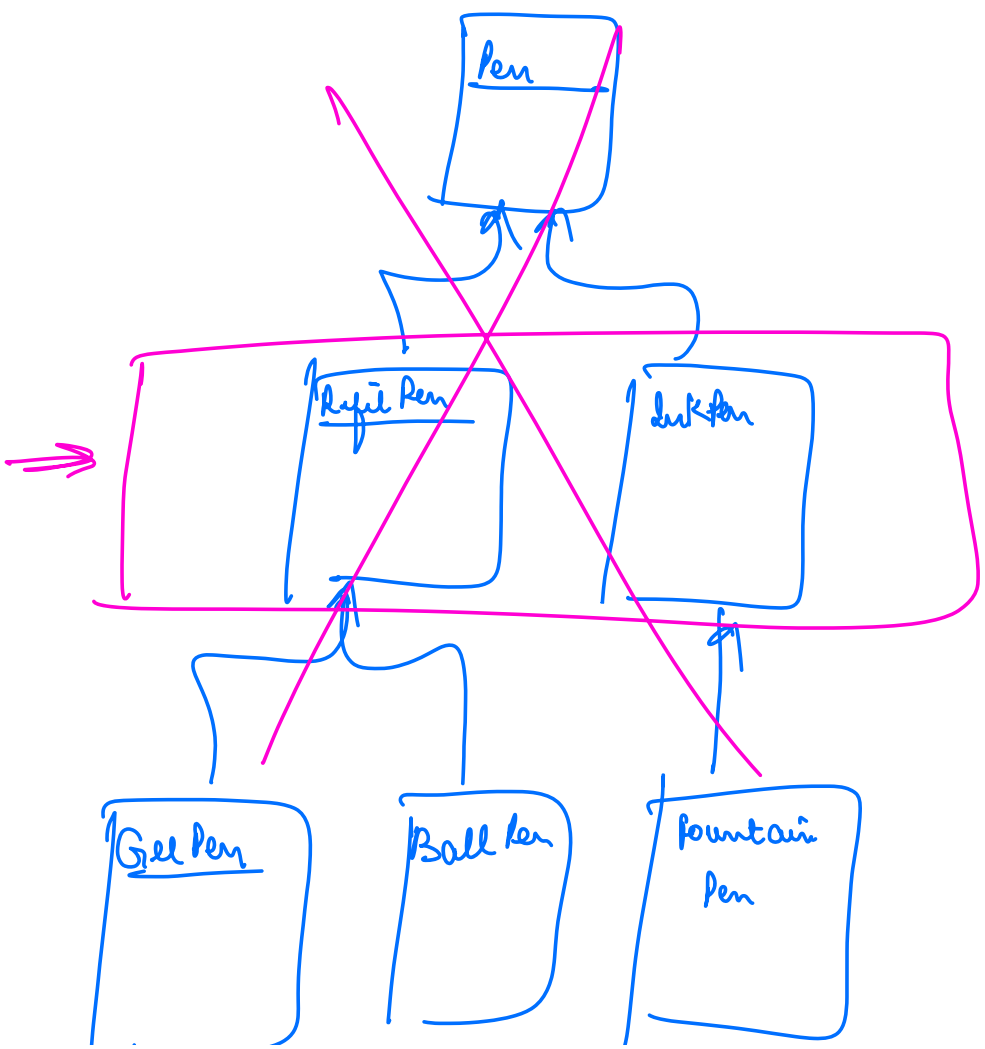
createPenOfType(PenType type) {
if (type == PEN) {

}

}

```
Pen g = new GelPen(CLICK, "Reynolds", "Add Gel");
```

Break till 10:~~22~~PM ⇒ Compo v/s Agg



→ every argument has 3 sides, 1 that first person says, 2nd that 2nd person and 3rd that is really

Pen (isCap, isClick, name)

```
Pen p = new Pen (false, true, "Reynolds")
```

Class Diagram

⇒ (i) to rep diff type of entities in a s/w system

⇒ (ii) to rep relⁿ b/w those entities

How to rep relⁿ b/w entities

2 Categories of relⁿ b/w entities :

→ (i) Inheritance (IS-A)

→ extending a class

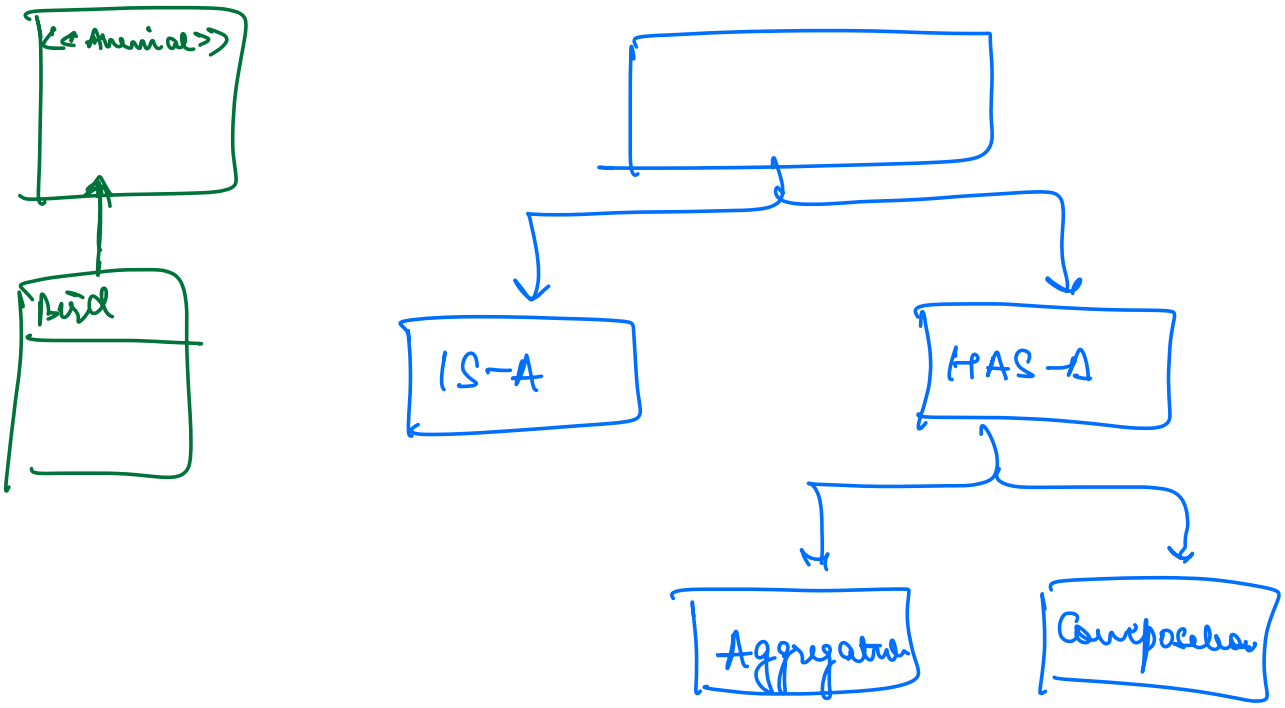
→ implementing an interface.

→ (ii) Association (HAS-A)

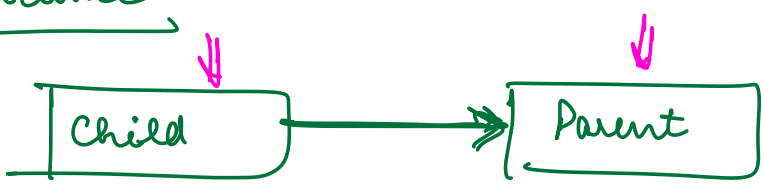
→ have an attr of another class.

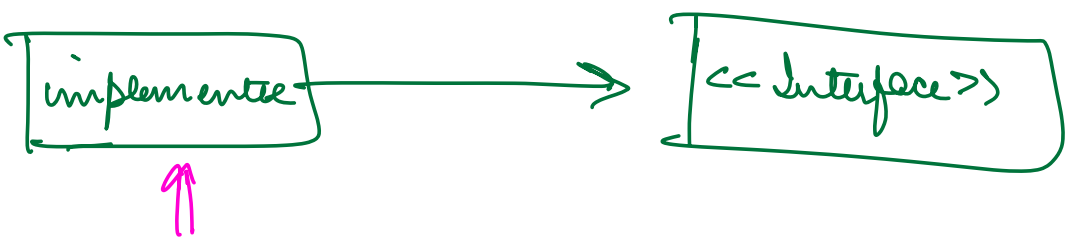
→ 2 subtypes

(i) Composition (ii) Aggregation



Inheritance





Association

① Composition

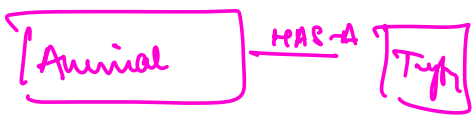
② Aggregation

Class Animal {

Type .

3

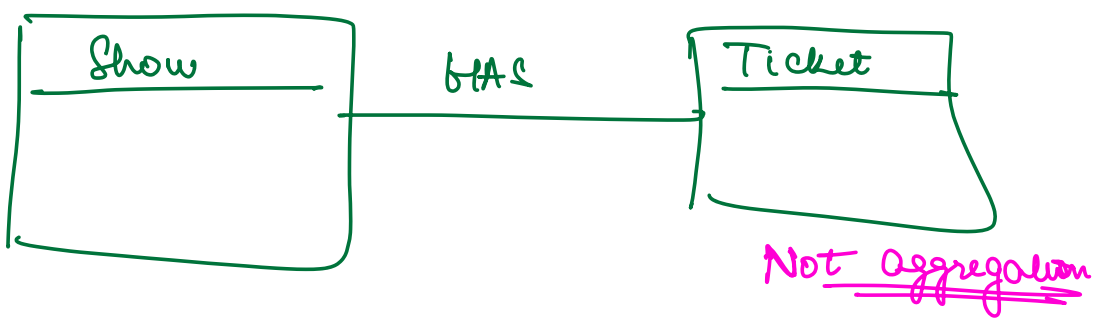
Aggregation



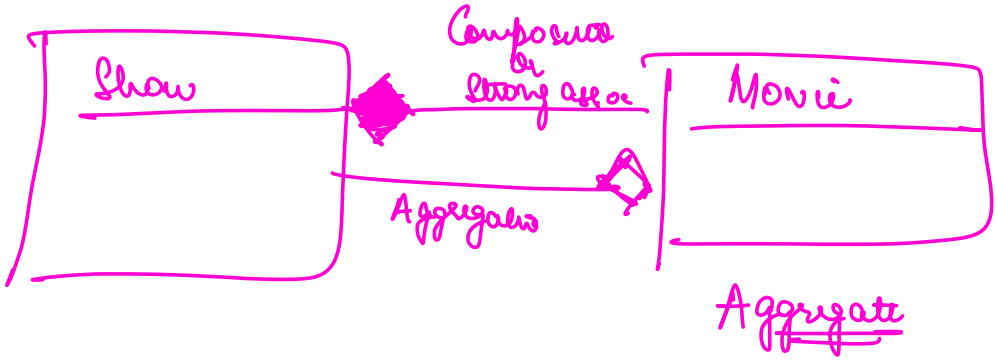
↳ collect / contain / assemble.

↳ the other entity has an independent existence

↳ the object of entity has a well-defined meaning w/o you.



Not Aggregation



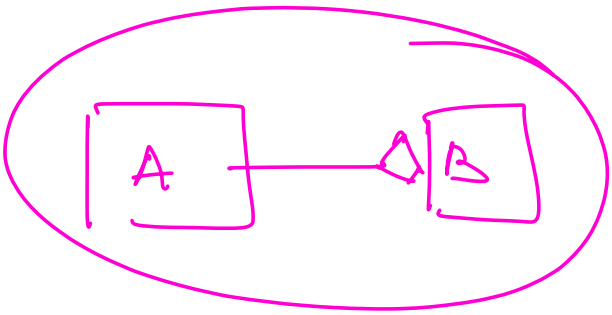
Aggregate

↳ also known as "weak association"

Class A {

B b;

}



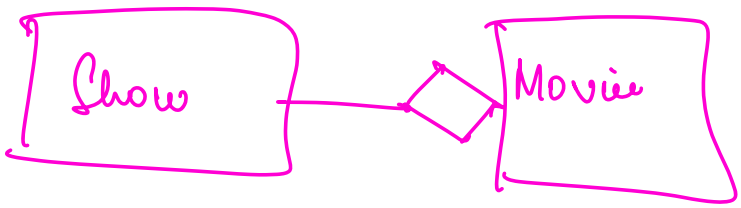
A has B

⇒ but A is not needed for B to exist

⇒ A is just collecting B.

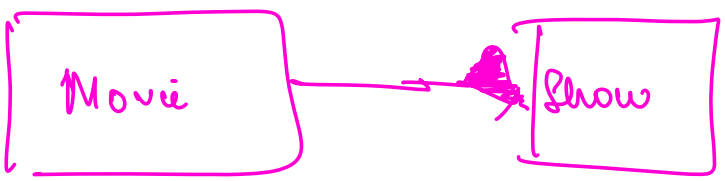
⇒ Objects of B have independent existence.

```
class Show {
    Movie m;
}
```



Aggregation

```
class Movie {
    ~~List<Show>~~ Show
    name
    title
    Actor.
}
```



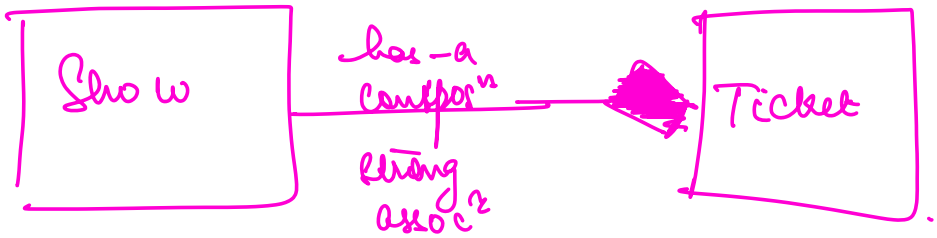
Composition

Composition

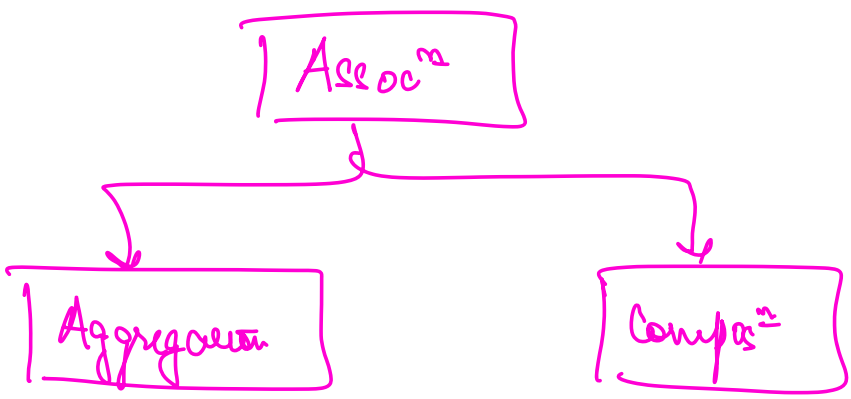
↳ maker / creator

↳ Strong Assoc^{n}

↳ A composes B when A has-a B
And objects of B don't have any well defined meaning w/o corresponding object of A



Class Show {
    List <Ticket>

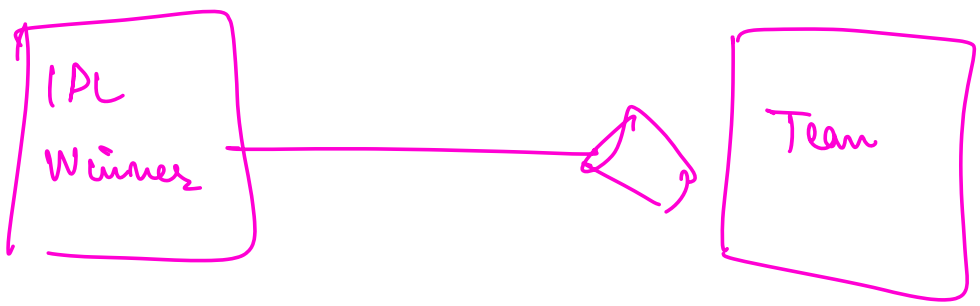


draw.io

weak assocn / Aggregation

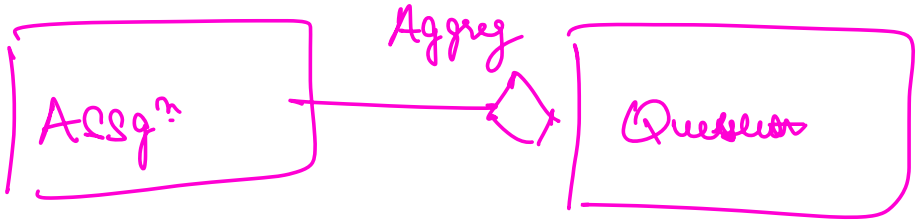
strong assocn / Composition

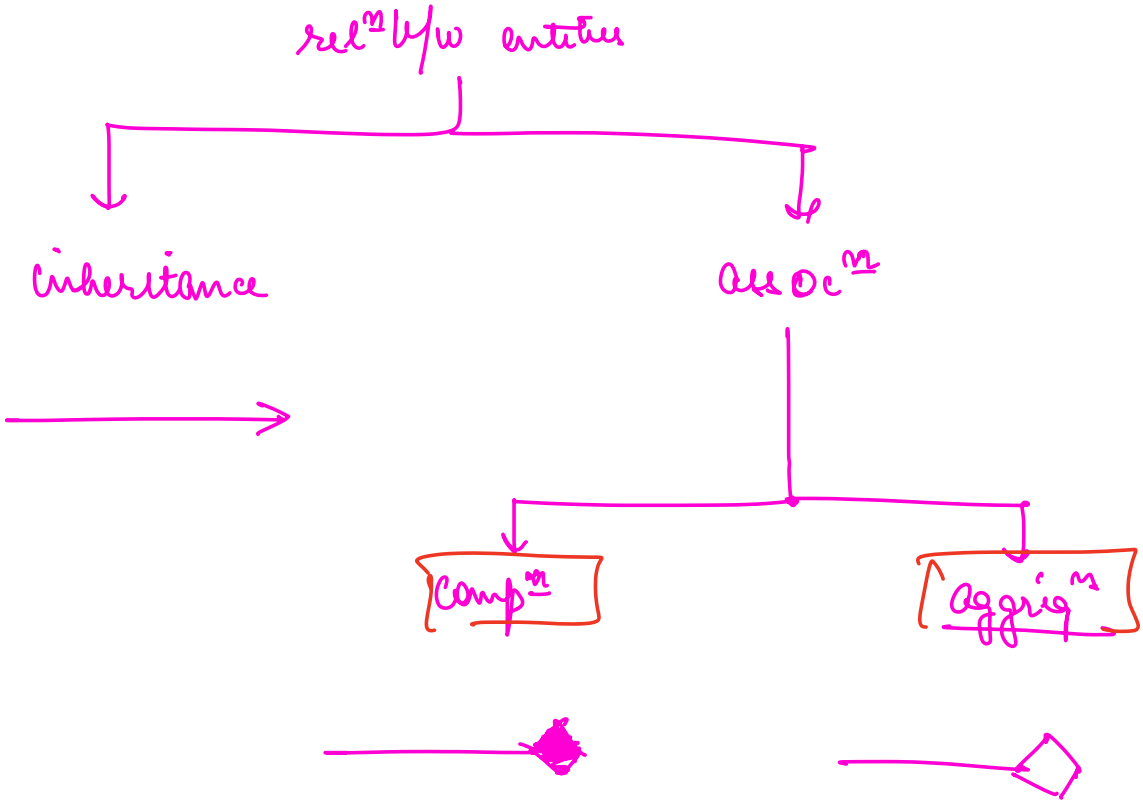

Movie
Aggregation
Actor


IPL
Winner
Team

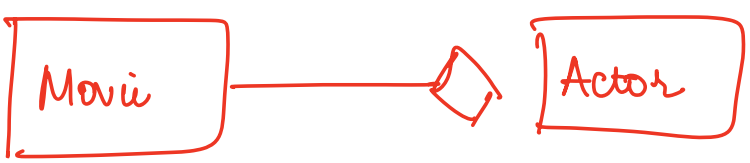
ACB


Class
Comp
Assgn


Assgn
Aggreg
Question



→ In terms of code they are same.

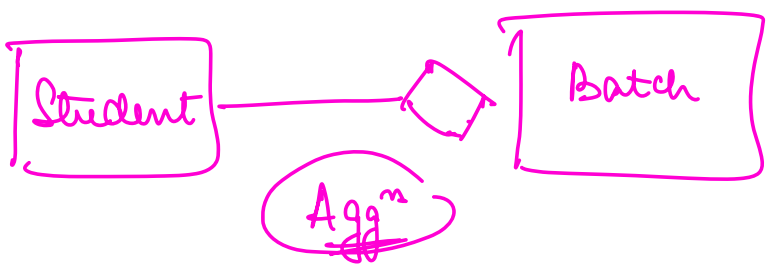


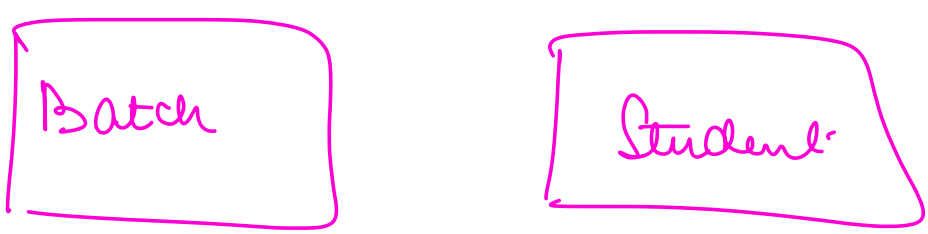
A has a B

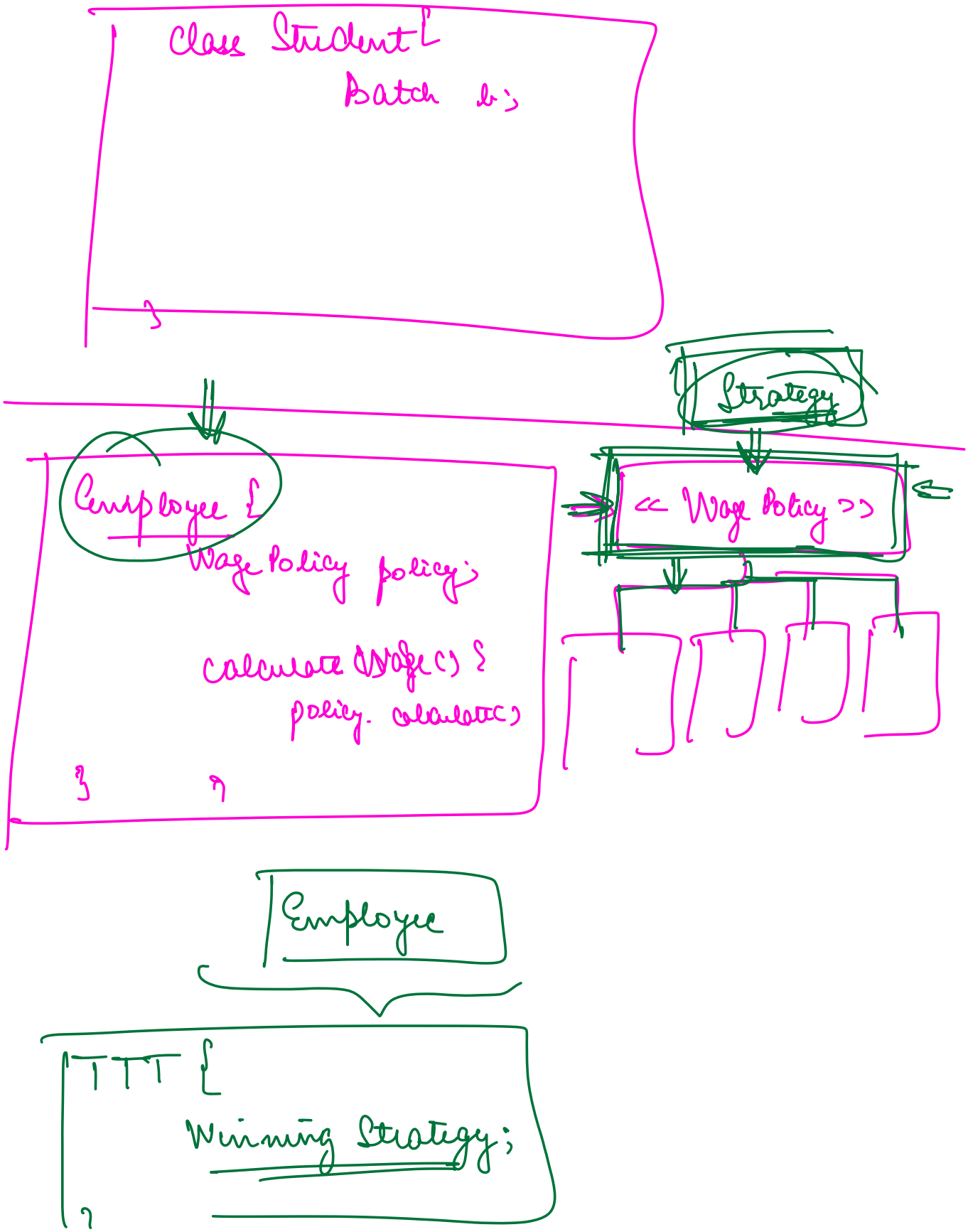
if B can exist w/o A → Agg

— Can't —— → Comp^n


```
Actor {
    List<Movie>

}
```






Class Student {
Batch b;
}
Strategy
Employee {
Wage Policy policy;
calculate Wage() {
policy. calculate()
}
}
<< Wage Policy >>
Employee
TTT {
Winning Strategy;
}