[=> UML Diagram
=> class diagram]

=> How to represent LLD of a system

=> Communicate

- Manager → work, approval, appraisal
- Team Lead | Architect → system design [LLD + HLD]
- QA → requirements
- Business (CEO, PM) → business requirements
- Clients → business / product requirements

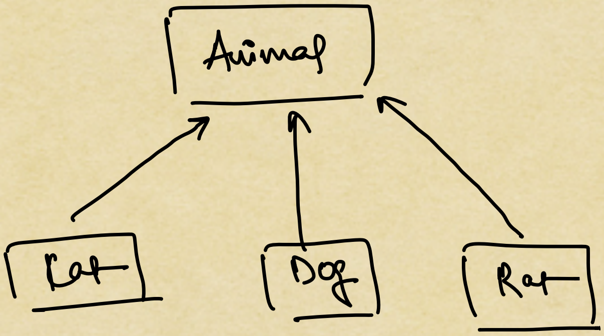
Ways to communicate:

i) words → email / slack / meetings
↓
leads to ambiguity
↓
Better [Pictures / Diagrams / Images / FlowCharts]
↳ less Ambiguity / easy to understand / visualization

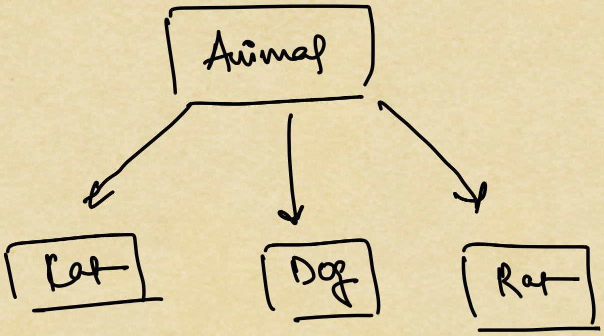
If there is no standardization of pictures/ flowcharts/ diagrams => confusing / time taking

Inheritance

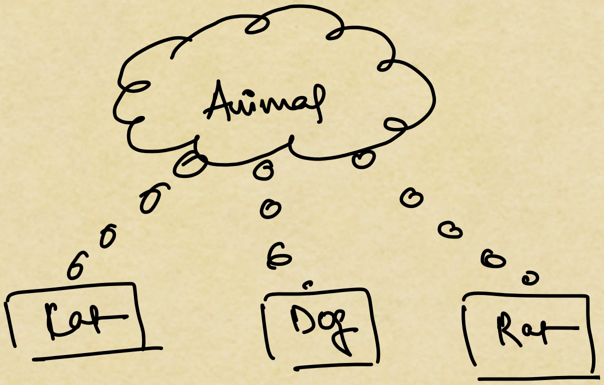
India



Germany

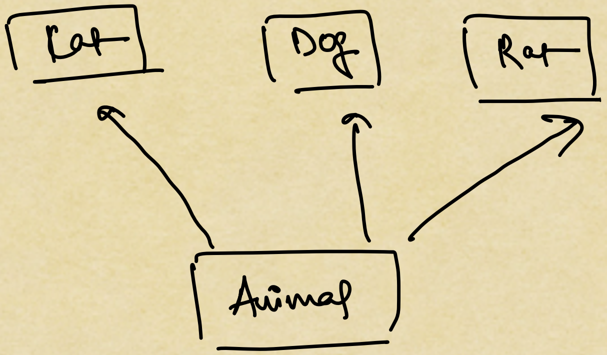


USA



} => Raising cats & dogs

UML



⇒ UML Diagrams

UML ⇒ Unified Modeling language

↓

Standardization on how to represent different SWE concepts in diagram

⇒ Types of UML diagram:

i) Structural ⇒ how the codebase is structured

ii) Behavioral ⇒ how the system works,
flow of the system

Structural	Behavioral
i) Class Diagram	i) Activity Diagram (HW)
ii) Package Diagram	ii) UseCase Diagram
iii) Object Diagram	iii) Sequence Diagram *** (HW)
iv) Component Diagram (HW)	

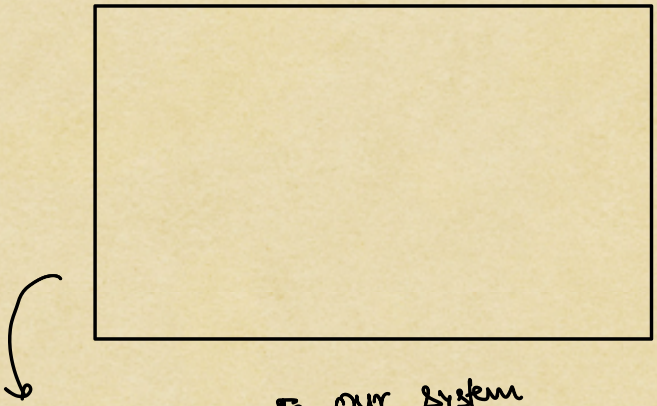
=> **Use Case Diagram :-**

→ features/ functionalities being used by our system

→ who are using those features/ func.

=> **5 major key words:-**

=> **System Boundary**



Rectangle represents our system

1P things => Inside rectn

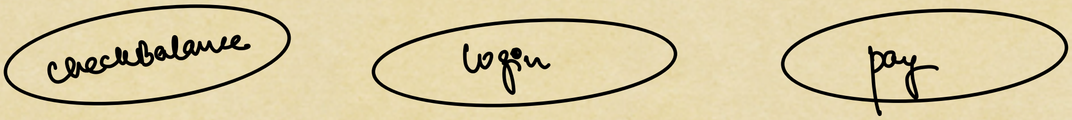
3P things => outside rectn

=> **Use Case**

=> functionality and features

=> Must always be VERBS

=> Represent by an Oval

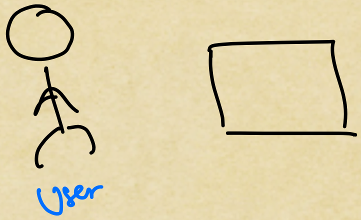


⇒ Actor

⇒ people who are using particular use case

⇒ must be nouns.

⇒ represent by stick diagram

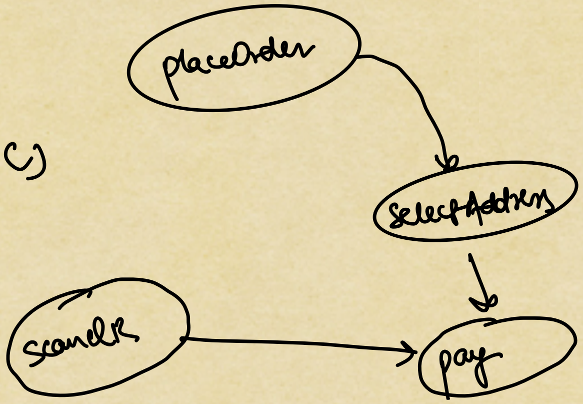


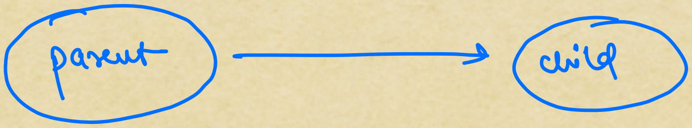
⇒ Includes

⇒ overall flow of a feature

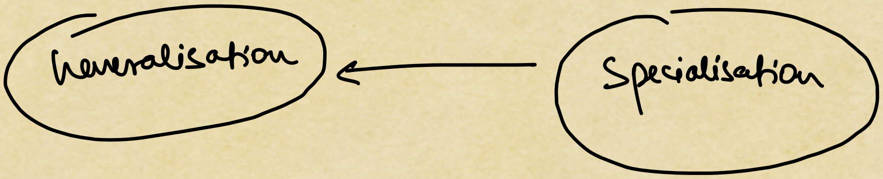
⇒ parent use case includes child use case

```
placeOrder(){
        selectAddress()
        pay()
}
```

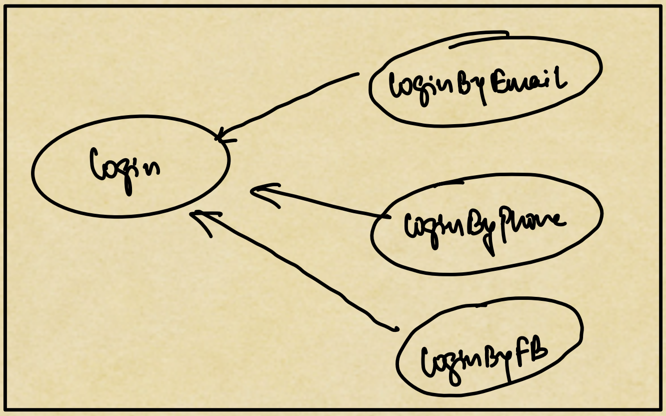


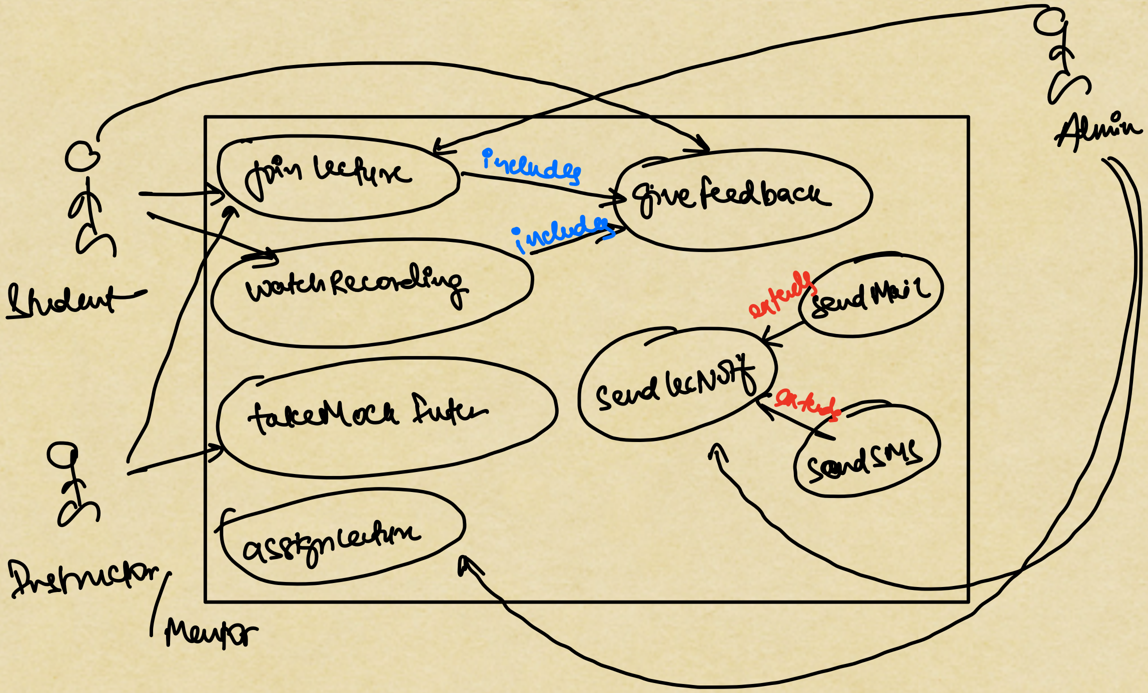
⇒ 

⇒ <u>Extends</u>



⇒ if a feature has multiple variants





⇒ Draw usecase diagram :-

⇒ Ecommerce appⁿ

⇒ 5 use cases

⇒ 2 actors

⇒ 2 use cases ⇒ includes

⇒ 2 use use ⇒ extends.

⇒ Class Diagram

⇒ representing diff entities in our system

→ Class

→ Abstract Class

→ Interface

→ Enums

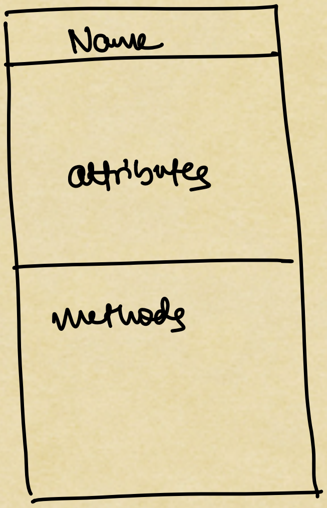
⇒ representing relationship b/w the entities

→ implⁿ of interface

→ extends a class

→ association & composition

⇒ Class



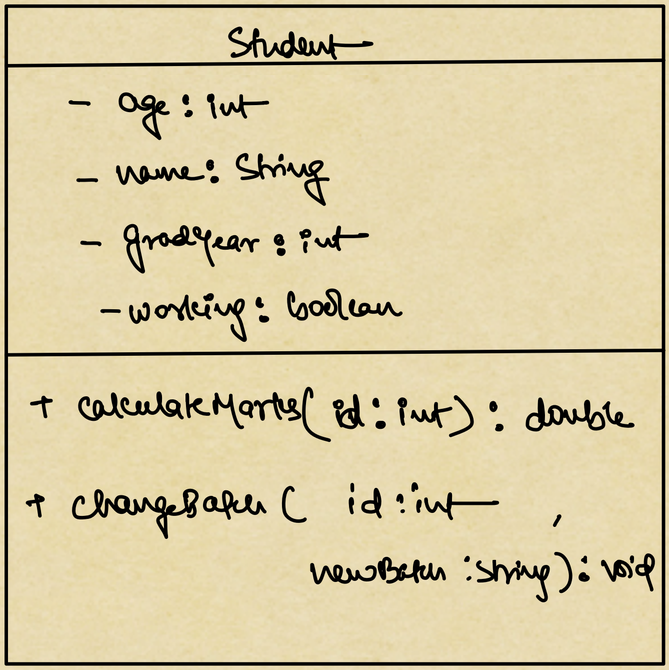
access modifier | name | : | datatype

⇒ ⇓

- public +
- private −
- protected #
- default

⇓

access modifier | name | (parameter [attributes]) : | return type



Student

- age : int
- name: String
- gradYear : int
- working: boolean

+ calculateMarks(id: int) : double

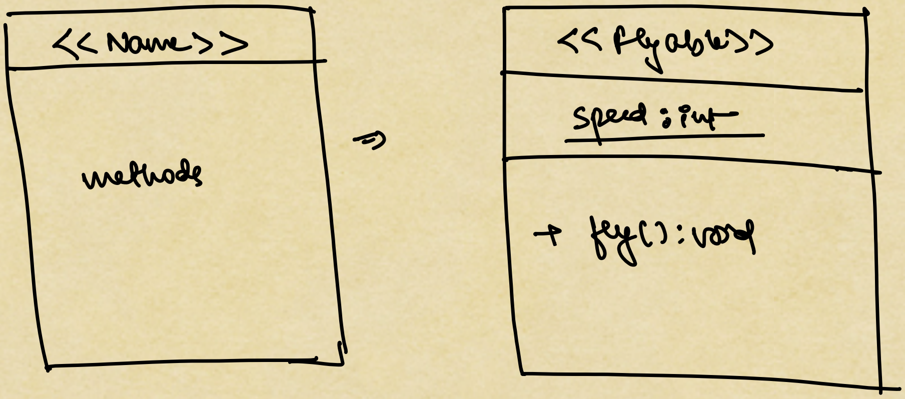
+ changeBatch (id : int ,
newBatch :String): void

```
class Student {
        private int age;
        private String name;

        public double calculateMarks(int id){

         public void changeBatch(int id,
                                   String newBatch){
              }
        }
}
```

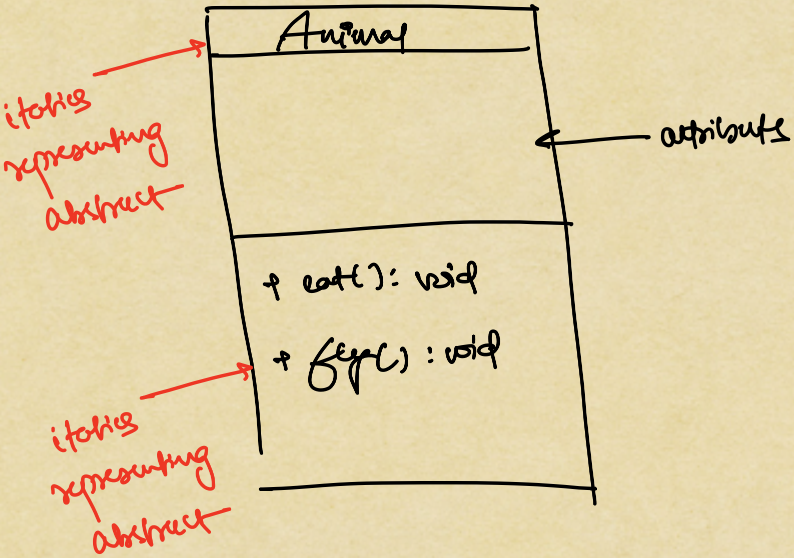
⇒ Interface



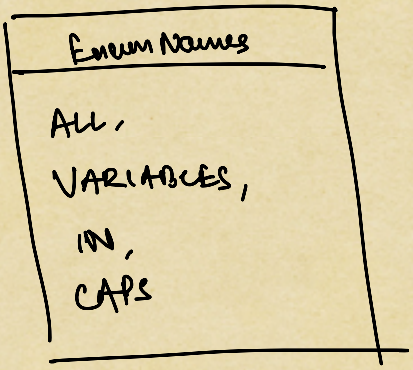
* Static variables/methods are represented by an underline on them

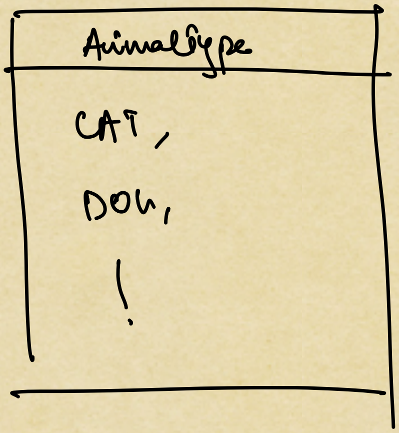
(anywhere)

* Abstract Classes

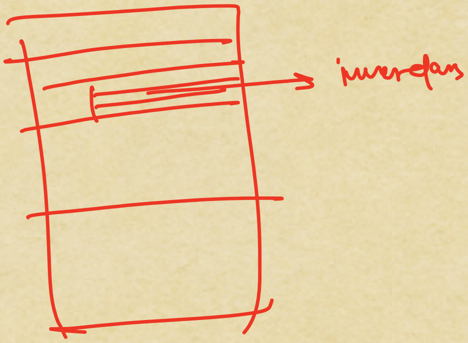


* ENUMS





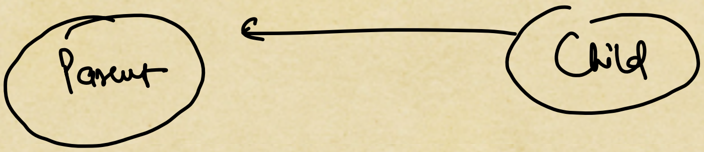
FINAL => BOLD

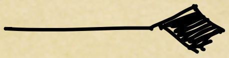
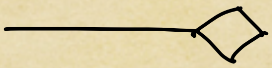


=> Represent relationship b/w entities:-

2 types
- IS A [Inheritance] (extends, implements)
- HAS A (having attribute of another class)

IS A (Parent Child relⁿ)



HAS A (Association relⁿ)
- Composition 
- Aggregation 

⇒ Aggregation

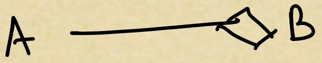
A has B

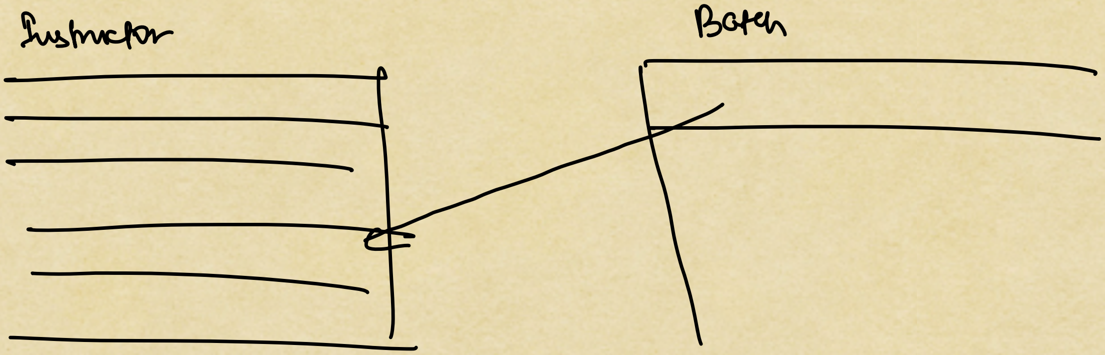
or, B has A

A has a B

↓

existence of both of them is independent of each other

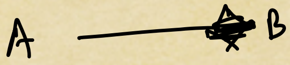




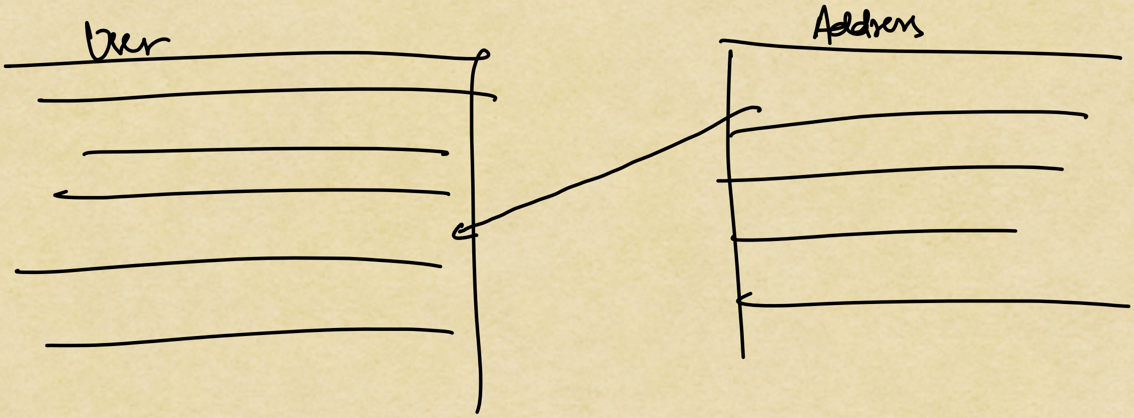
Instructor has a batch

⇒ Composition

A Has a B



existence of any one entity depends on other



Address

↓

SSM

FL

HO

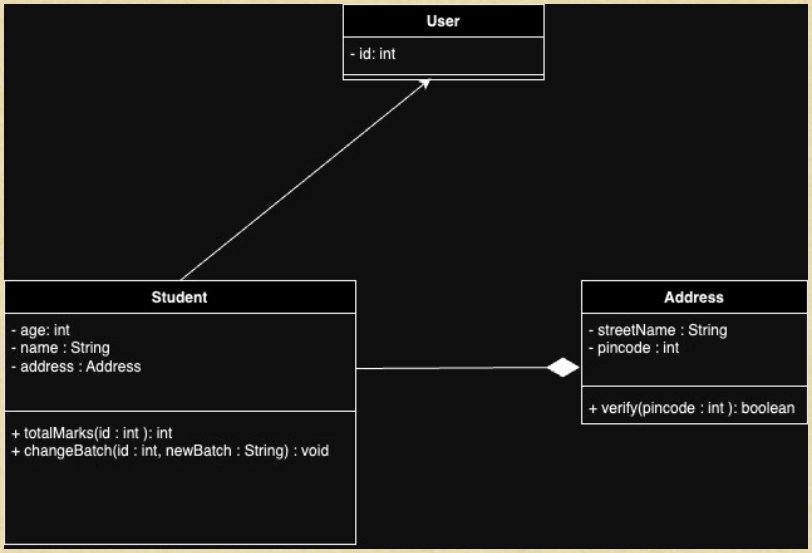
Str

Loc

Pin

code

phone



User

- id: int

Student

- age: int
- name : String
- address : Address

+ totalMarks(id : int): int
+ changeBatch(id : int, newBatch : String) : void

Address

- streetName : String
- pincode : int

+ verify(pincode : int): boolean