Software Design & Analysis.

LEC 1

Software Development Life Cycle:

- Requirement / Analysis
- Planning
- Design
- Implementation
- Testing
- Deployment / Installation
- Maintenance

LEC 2

Characteristics of Good Programs:

- Readability (easy to read)
- Writability (easy to write)
- Reliability → good error handling
- Reusability
- Modularity (→ create smaller components)
- Abstraction
- Maintainability (easy to change)

e.g.:- If you have used a fun. at it's easy to change. But if you have used code instead it will be difficult to change code everywhere

Reliability Example:

```
cin >> a;
if (a==10)
{ cout << "10"; }
```

(1) e will not give error. Java will. So, Java is more reliable.

Because in C in if both int & bool works. Since 10 will be assigned to a and has int value. So, print statement will always work. In Java, it will not work.

(2) In Java, you don't need to do deallocation there is automatic garbage collector

Date: Day:

Selection sort:

5 2 4 1 3

```
void function ( int array [ ] ).
{   for (int index=0; index < array.size()-1; index++)
    {   int smallest = array[index];
        int p = 0;
        for (int i=index+1; i < array.size(); i++)
        {
            if (smallest > array[i]),
                smallest = array[i];
                p = i;
        int temp = array[index];
        array[index] = smallest;
        array[p] = temp;
    }
}
```

5 2 4 1 3

1 2 4 5 3

1 2 4 5 3

1 2 3 5 4.

1 2 3 4 5

more readable, reliable, reusable

```
void sel_sort (int a[], int n)
{   for (i=0; i<n; ++i) {
        int m = min (a, i, n)
        swap (a[i], a[m])
    }
}
```

create function to increase readability

Abstraction :-

- Concept with which we can work without bothering about the underlying details

e.g :
function, classes

- Good abstraction hides implementation details and provide easy to use interface

e.g. Cars

Maintainability :

```
a= 2*b*5.

fun(x,y,5)
```

- If you want to change value from 5. you will need to change 5 everywhere instead it's better to create a constant

```
const int tax =5
```

:Lec3:-

Inheritance & Polymorphism

Inheritance :
Process by which properties (data members, functions) of parent class are inherited by child class

"Polymorphism" means many-form.
When a parent class reference is used to refer to child class object.

Date: Day:

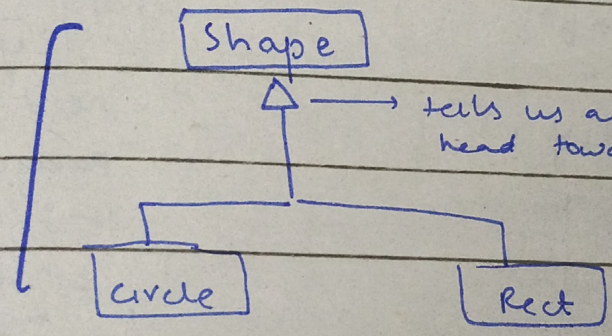
same line of code may invoke different functions at each instance

- void update (circle* a[ ], int m, Rect* b[ ], int n)

```
{
    for( int i=0 ; i<m ; ++i )
    { a[i]->draw( ); }

    for ( int j=0 ; j<m ; ++j)
    { b[j]->draw( ); }
}
```

UML class diagram



- void update (shape *a[ ], int n)

```
{ for (int i=0 ; i<n ; ++i )
    a[i]->draw( );
}
```

→ abstraction. All working is hidden, and fun of all classes are called from one array.

- Place pointers of circle & rect. in array of shape then we will only need one loop.

a class can call constructor of any other class and store the pointer and invoke funs of that class

- Static binding without virtual keyword.
- Dynamic " when " " is used

In following code:

In dynamic binding, the function of type of pointer is run. (stored in array). If in shape array, circle pointer is place, fun of circle is used. In static, always fun of shape will be called.

```
int main () {
shape * b[M] = [
b[0] = new circle (....) ;
b[1] = new rectangle ();
" " "
" "
];
update (b, M);
```

Circle c
Rect r;
b[0] = &c;
b[1] = &r;

returns pointer to class + allocate memory in heap

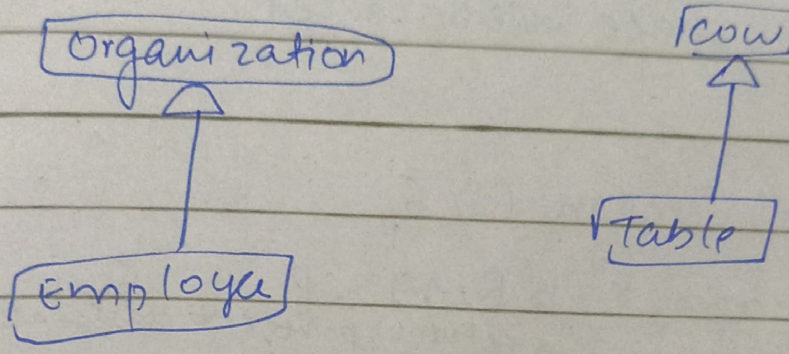
- Good Maintainability
- " Scalability
  - Addition of new feature is easy

IS-A Rule:

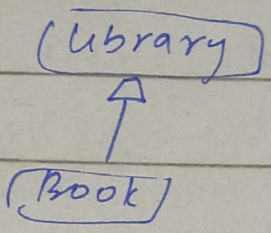
- "Child class is a parent class."
- Evaluate the statement, if it is correct than 'is a' relation is correct.
- ✓ Circle is a shape

Date: Day:

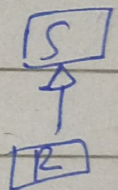
- Table is a cow ✗
- Employee is a organization ✗
- Table has four legs, but we can't inherit it from cow, since cow also has four legs



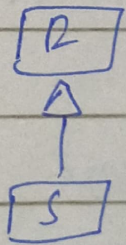
- Book is a library ✗



- Rectangle is a square ✗
- Square is a rectangle ✓

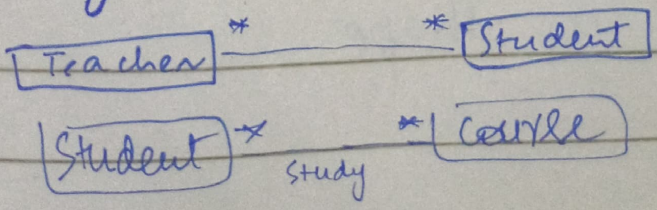


Square inherits rectangle -



(LEC 4) —

Object Assosiation:

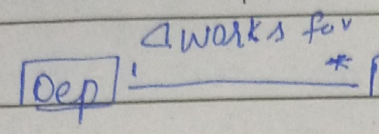


class Name
↓
Noun
Assosiation Name
↓
verb.

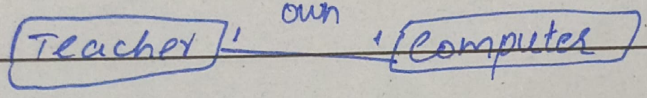
inheritance → ↑ ↑ → association

Date _____ Page _____

Multiplicity / cardinality : 1 — 1, M — 1, M — M



- Normally, we read from left to right. If we want to read reverse we can write a arrow




```
class teacher {
  //  //
computer *com;
  //  //
  };
Class computer {
  //  //
Teacher* t;
  //  //
};
```

one to one

```
Class dept {
Employee *emp[N];
  //  //
  };
Class Employee {
dept *dname;
  //  //  };
```

many to one / one to many.

Day.

→ unidirectional association

—— bidirectional "

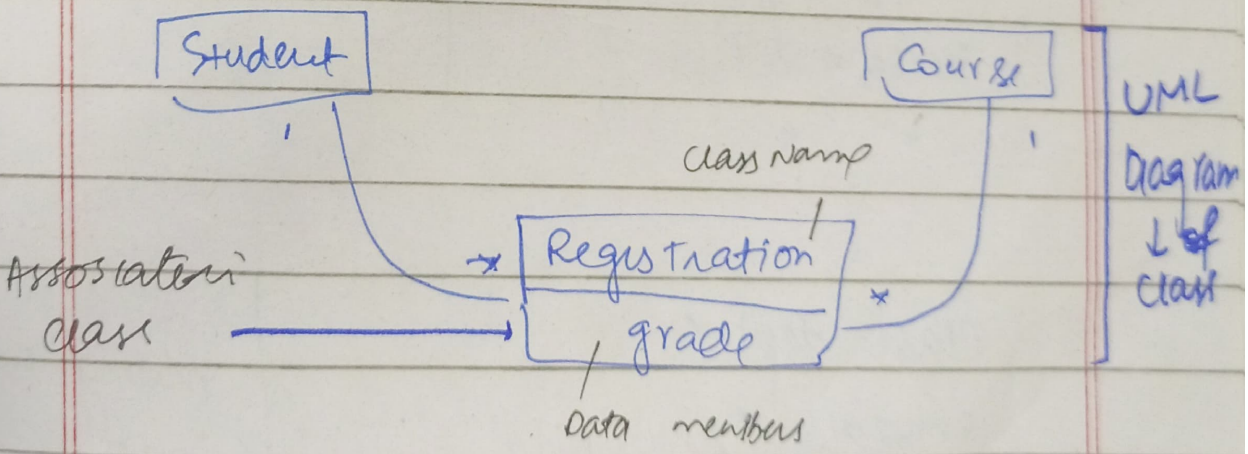
- Entity which has arrow towards it do not contains pointer of other entity

Teacher → Computer

Teacher will have pointer of computer

Whenever we have many to many association. we remove it and form two associations of one to many.

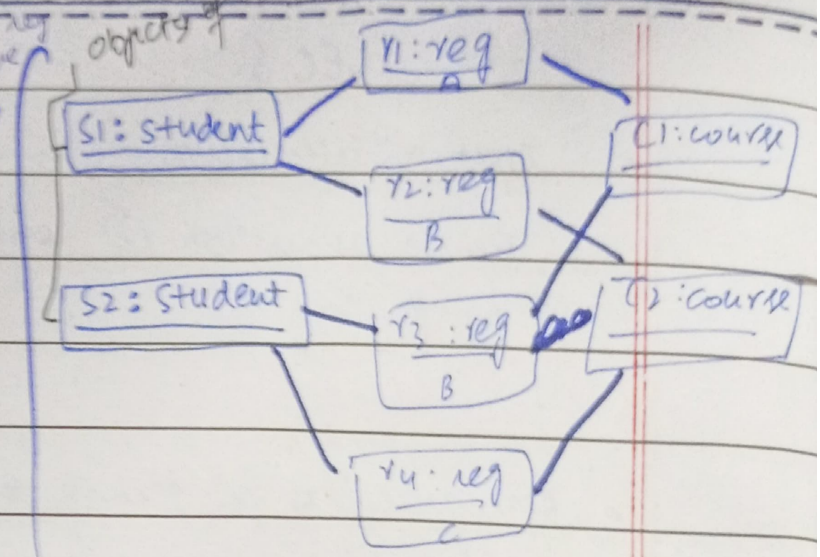
Student * —— * Course



- Registration has one pointer of course and student for one relation b/w them.
- Student & course will have many pointers of Registration.

If one student Registers in course. Array of pointer of reg type in student and course will be filled with respective info. Similary, for reg class we will have one pointer of student and reg which will also be filled. If student registers in more courses then will happen again

objects of Student

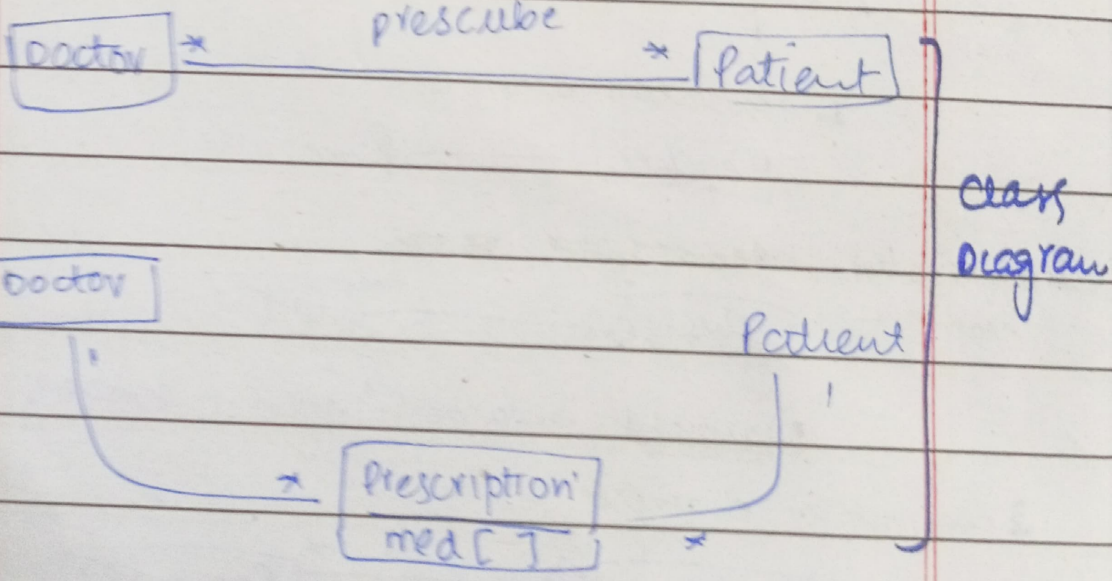


Object Diagram → comes under Use Diagram

- Code will have same name of objects as object diagram
- Registration will have same no of objects as total grade of Students (i.e 4)

Students have registered in total of 4 courses, so 4 grades will be awarded

Doctor–patient:



Associative class.

Date: Day:

LEC 5

Types of association:

(1) Simple association (2) Aggregation (3) Composition.

special types of association

- Aggregation is a strong kind of association
- Composition " " " " " aggregation

- Association → has a → Part-whole relation
  - one thing made up by other small things
- Aggregation → has many. → life-time binding
- composition → has many.

cart * ——— * wheel

But we can use any wheel we are not bound to use only one wheel

1. Association example:

Teacher ——— Computer

2. Aggregation example:

Vehicle ◇——— * Parts

- Diamond towards whole (i.e vehicle) side.

3. composition example

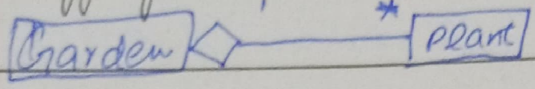
Building ◆——— * Room

- Filled diamond towards thing which can't exist independently

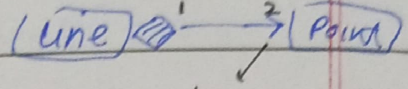
← Min. points required for line to exist are 2

Date _____ Date _____

Aggregation :-



Composition

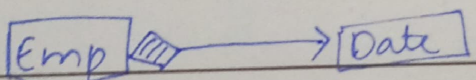


bcz composition is unidirection. no pointer of line in point

How to implement:

```
class Line {
private:
  Point p1, p2;
};
```

- There is always composition b/w class and its members



- ✗ Object in composition - in whole class of other class (i.e Line)
- ✗ Pointer in aggregation in whole class " "
- ✗ Pointers in association

- Class, data members are usually uni-direction

Imp:-

Pointers can also result in composition

To ensure pointer, can be used for composition pointer should exhibit object like properties. It should not exist independently, and can't be manipulated by outside.

Date: Day:

Composition using pointer:-

```
class Line {
private:
    Point *p1, *p2;
public:
~Line() {
    delete p1;
    delete p2;
}
```

Now, lifetime behavior also happen

Pointer can't exist without class

```
Line(int x, int y, int x1, int y1)
{
    p1 = new Point(x, y);
    p2 = new Point(x1, y1);
}
```

Do this or deep copy

```
Line(Point *q1, Point *q2)
{
    p1 = q1;
    p2 = q2;
}
```

Shallow copy

Don't do this

On this case, other classes can access it from main using q1, q2

Pointers outside of class can't manipulate it

Do deep copy for composition using copy constructor

```
p1 = new Point(&q1);
Point(Point *p)
{
    this->data = p->data
}
```

deep copy

Date _____ Page _____

```
int main() {
    Point a(10,20), b(80,90);
    Line l1(&a, &b);
    Circle c1(&a, 10);
```

- If we don't do deep copy a in c1 will start pointing to where a in l1 is pointing)x

Imp:

- If we don't do deep copy i.e "a" from main was stored in pointers which was inside the class. i.e p1 now, "a" which was also sent to c1 can be changed and this change (or change) will also "point to" pointer in class.

To avoid this manipulation, we do deep copy

Aggregation :-

Message Propagation :- → used in aggregation mainly.

- If a message is sent to whole and it transfer that message to parts, .

- only this small point, defer it from association

Its not necessary that it always happen

If we get price of car as whole at once

If total price of car is calculated by adding price of individual parts

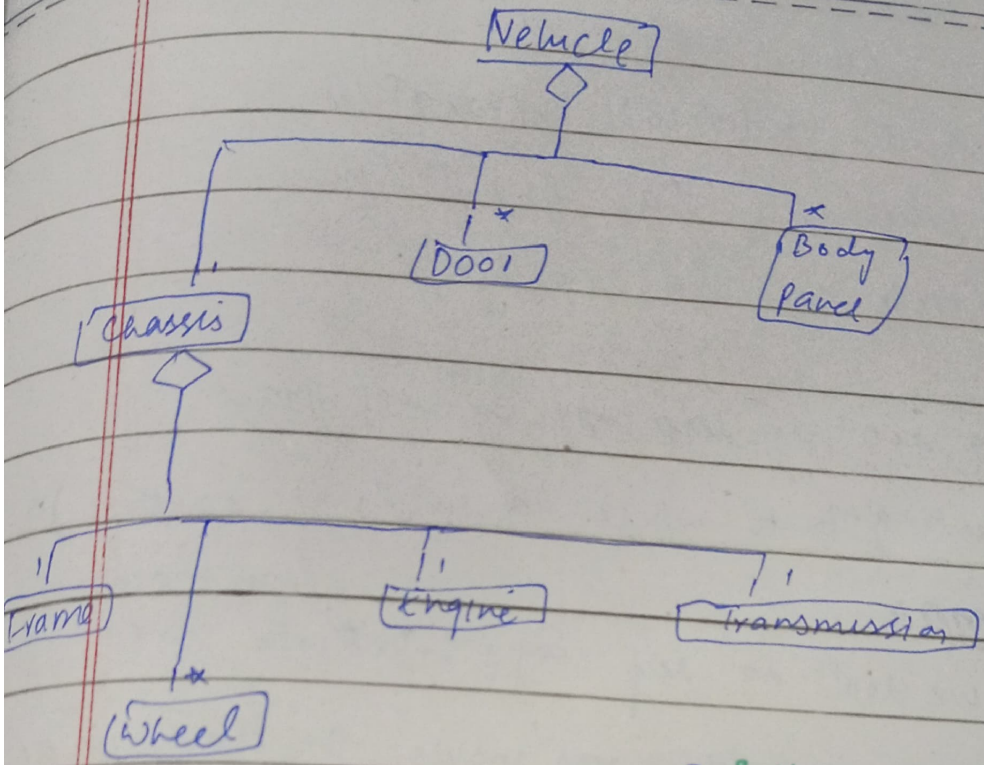
Do this or deep co

Shallow copy

Don't do this

Do deep copy fo composite using copy cons

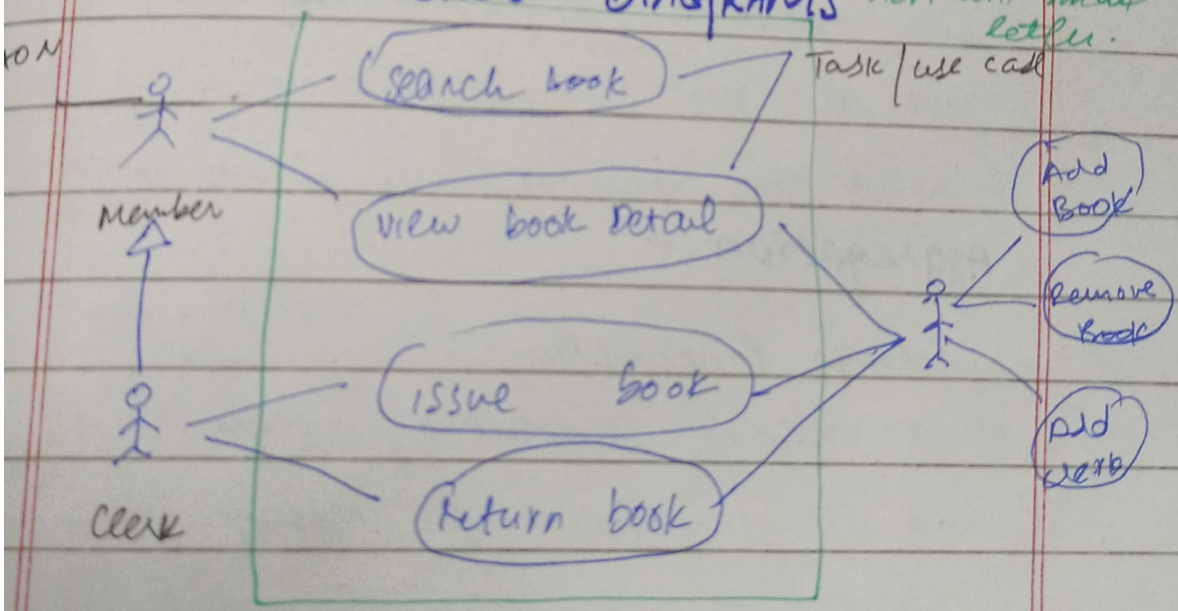
Date: Aggregation Hierarchy Day:



(LEC 6)

- System should be in rectangular box
- Use case name should start with small letter.

USE CASE DIAGRAMS

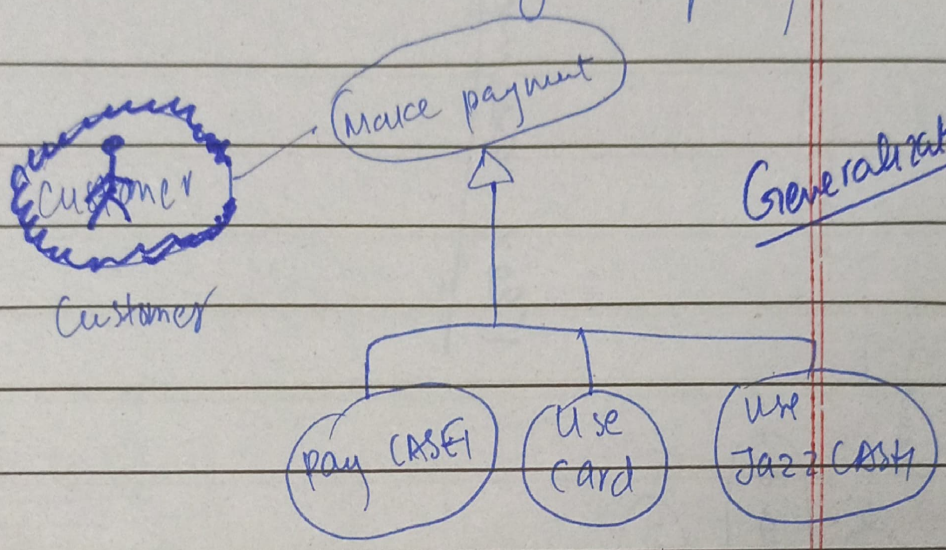


- Use-case diagram tells us about users and their purpose
- Can simple line or one with arrow (<→)
- One task does not necessarily

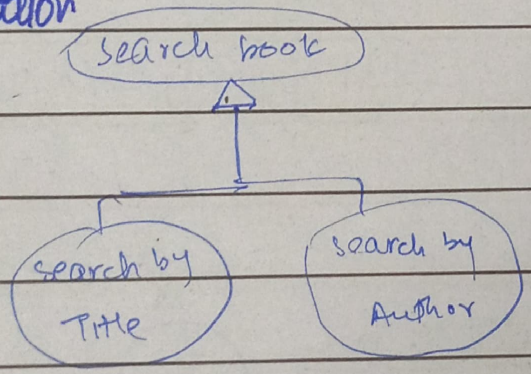
Users → Noun

Use cases → verb (first word if verb)

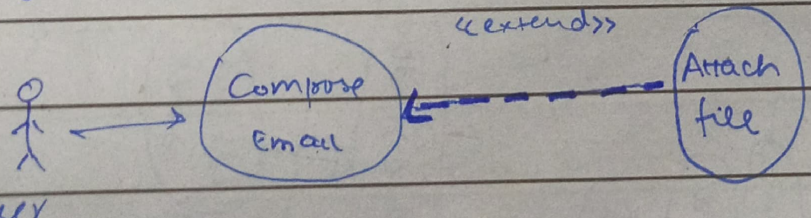
mean we will need one function for it. we could be using multiple functions.



Generalization



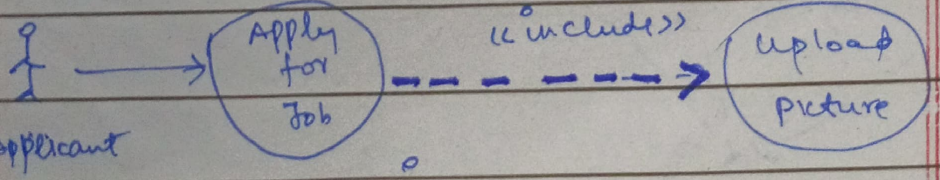
Extensions:



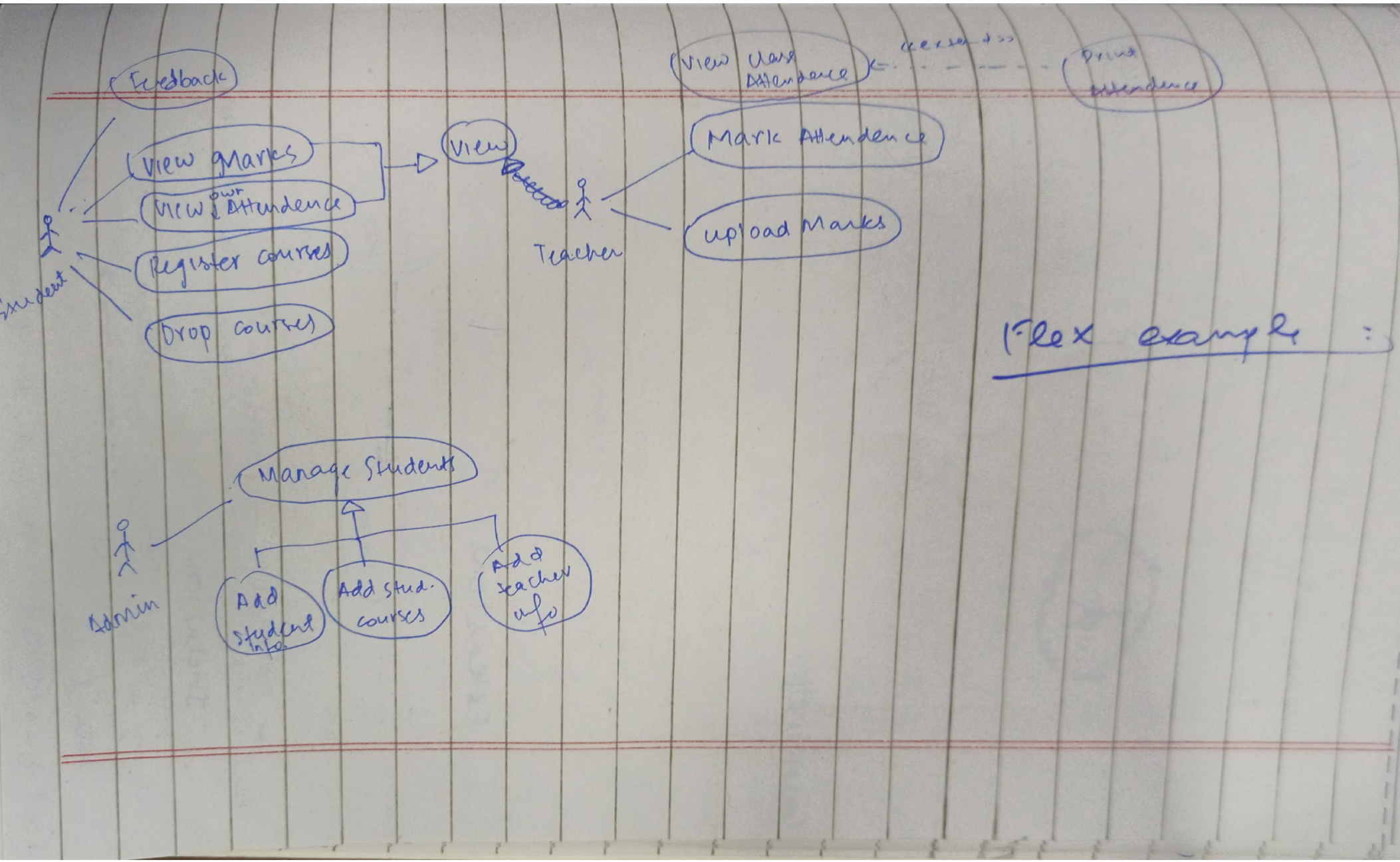
- Extension — optional

Read as → Attach file may extend compose email

Inclusion.



Read as → Apply for job, include upload picture



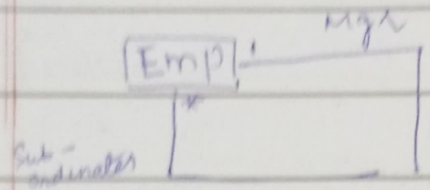
Flex example :

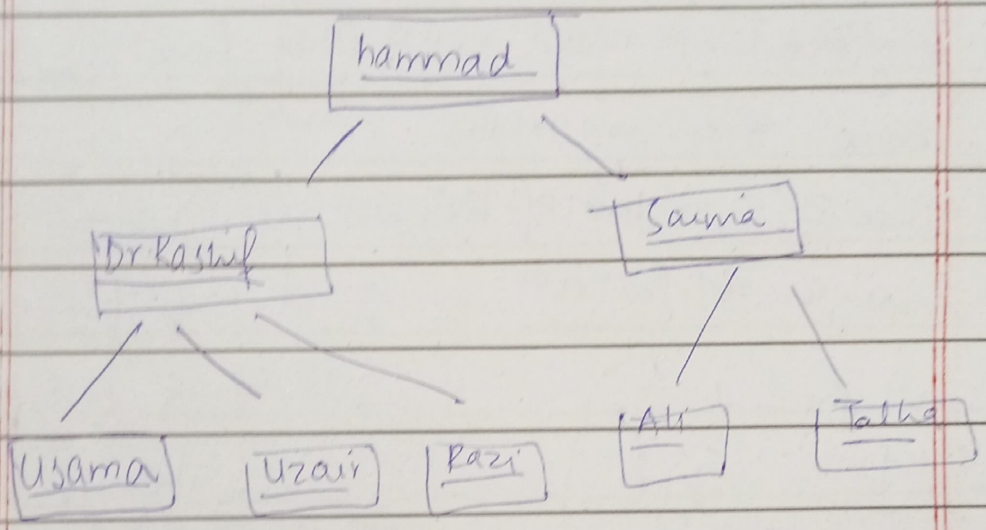
Date:

Day:

Date _____ Day _____

Reflexive Association..



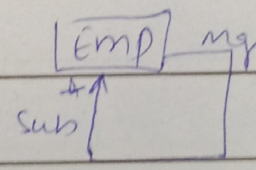



```
class Emp{
.....
Emp* mgr;
Emp* sub[N];
```

→ to point to manger of employee

→ point sub-ordinates of employee

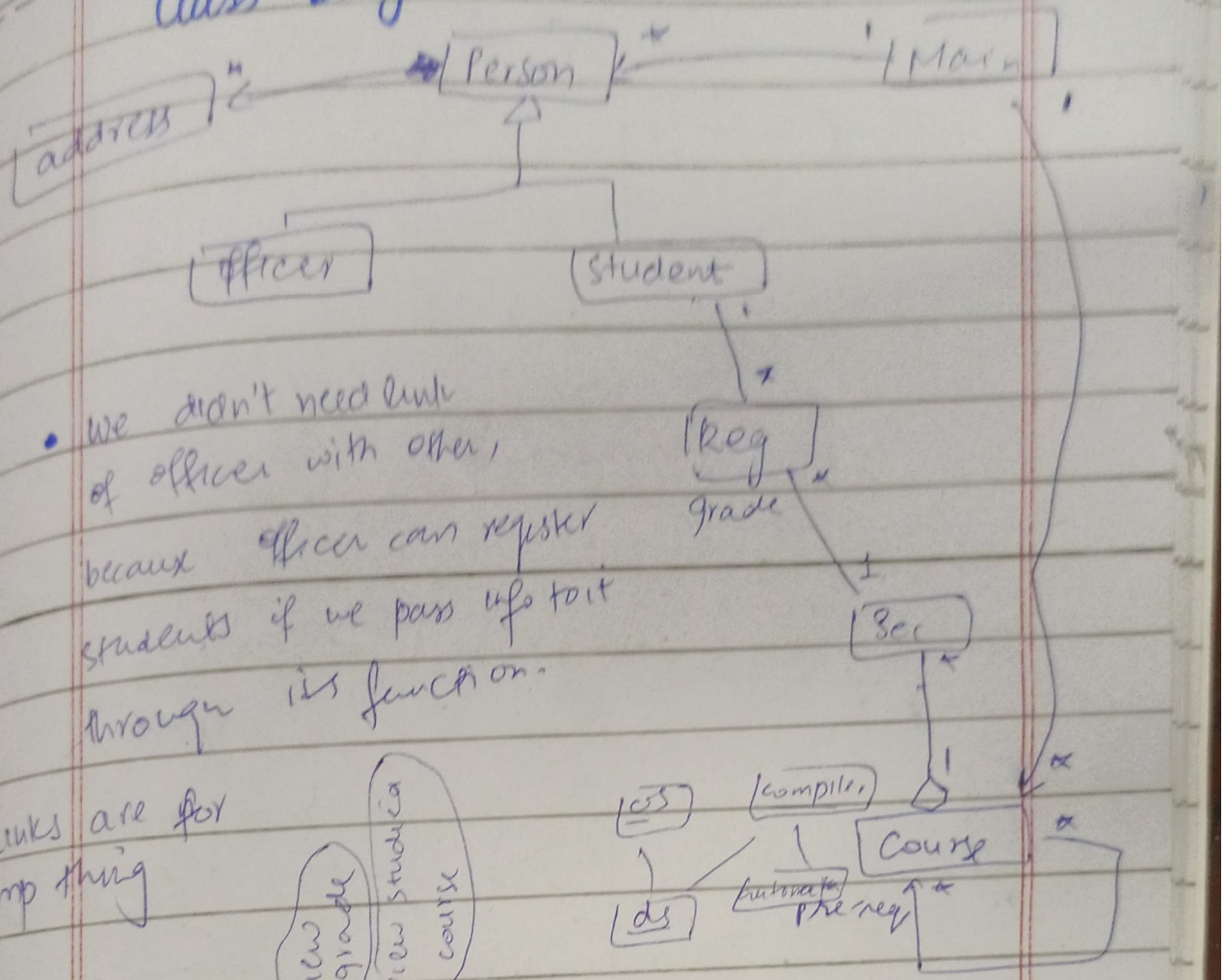
- if we have directional then we also have only Emp* sub[N];



Originally, student and section had many to many relation so we break it and created reg to store grades.

Date: Day: Thursday

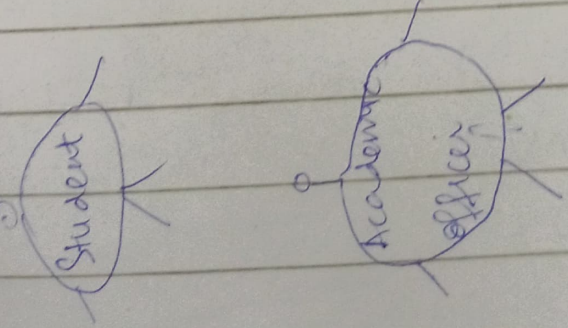
Class Diagram



- We didn't need link of officer with other, because officer can register students if we pass info to it through its function.

- Links are for imp thing

= Because we need section so, it can be accessed by student, so we don't use composition, we use aggregation



= Because we don't want address to be accessed anywhere else we use composition. Address will be as private member of person

Date _____ Day _____

* Direction & Multiplicity are diff things
  - pointers will be in which class
  - one / many

- Use case Diagram, Description
  - What to DO (Analysis)

- Class Diagram
  - How to DO

Activity Diagram:

- Fork and thread join work together.
  - creates threads.

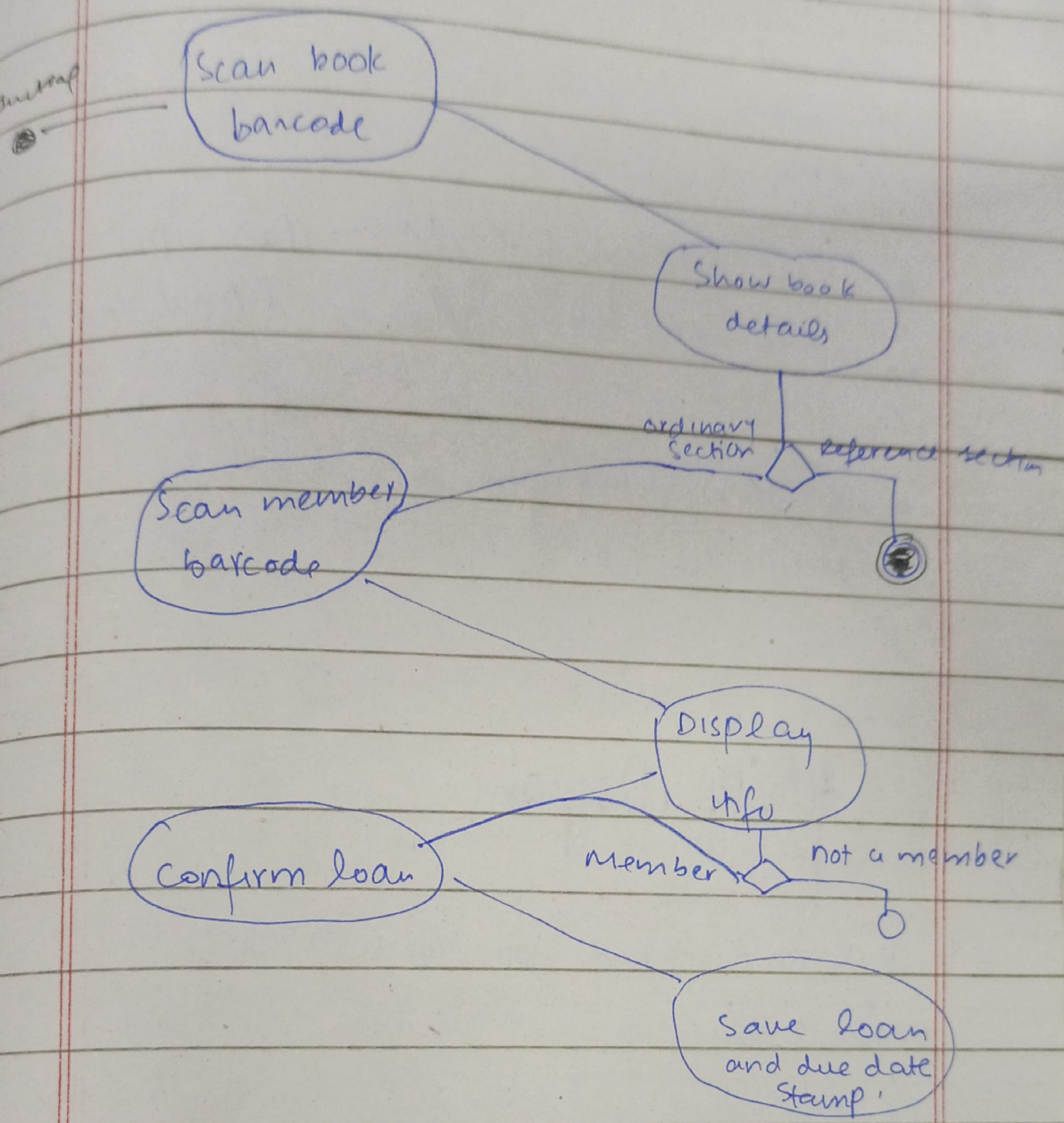
- Diamonds split or join

- If any thread reach final state all threads terminate

fork

diagram

join




```
void printCourse ( ) {
    for(int i=0; i<N; i++)
        a[i] → getCourse( );
}
void getCourse ( )
    c → getTitle ( );
void getTitle ( )
    cout << this → title;
```

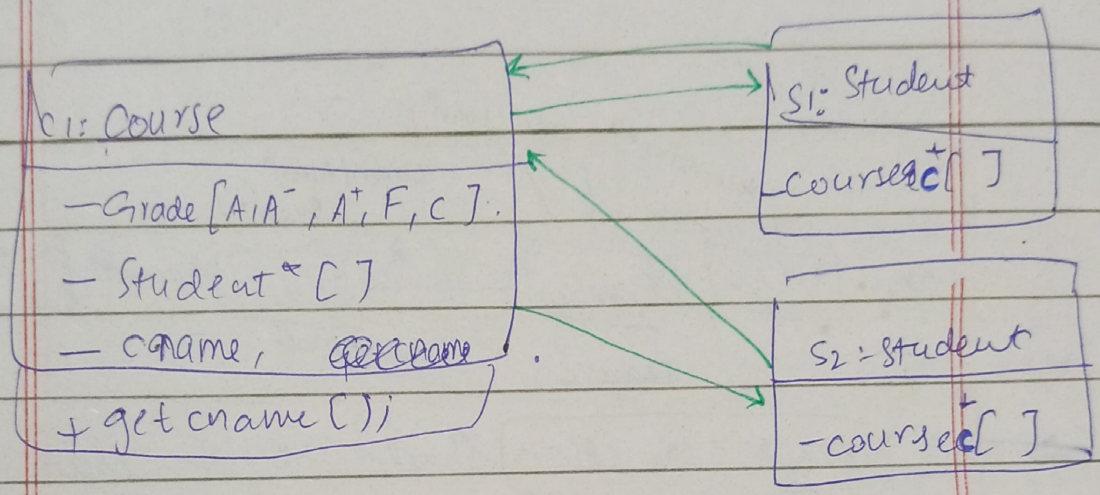
Mids incoming

Date: Day:

Many-Many without extra class Drawback

[Course] * ——— * [Student]

Let's say, we don't create another class for this and see what problem will we face



Drawback

- From this info, we see + course have grades array but B not able to tell which student has which grade
- We can't get grade of a student in course by this `S1→c[0]→grade`

If S1 is object

S1.c[0] → grade. → because grade is array and we don't know grade of S1 is at which index of array.

Valid calls :-

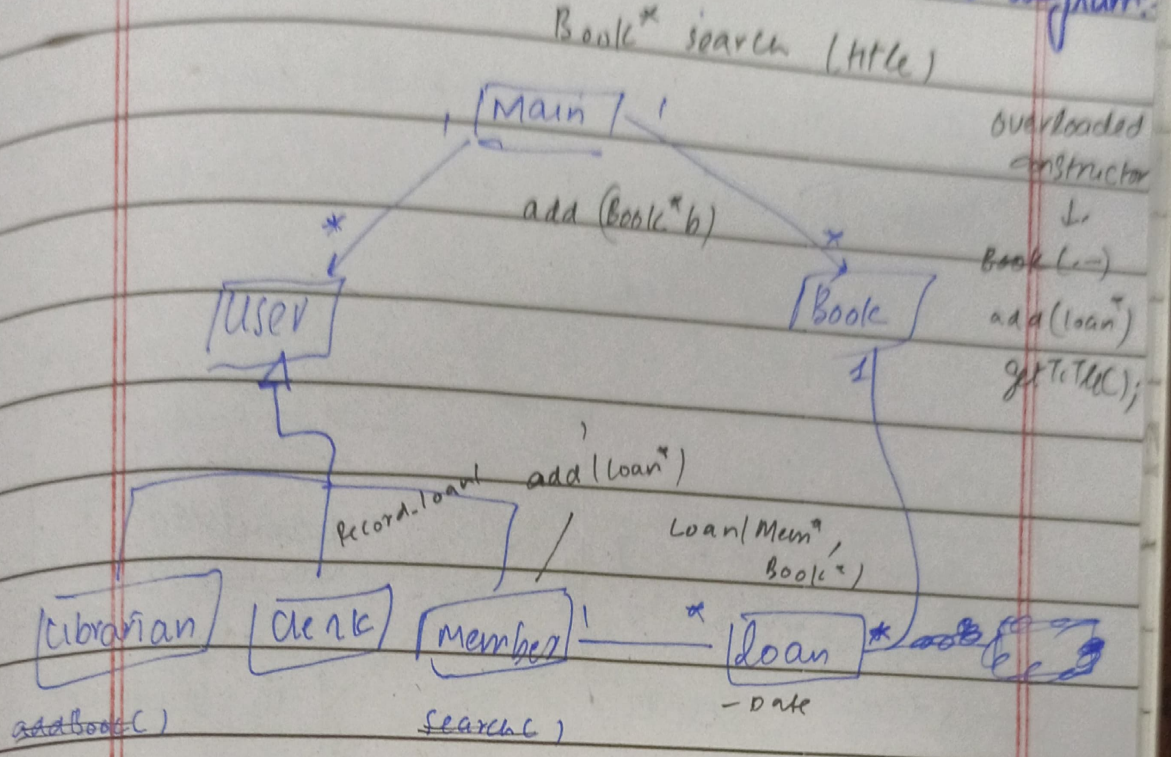
S1 pointer: S1→c[0]→getcname();

S1 object: S1.c[0]→getcname();

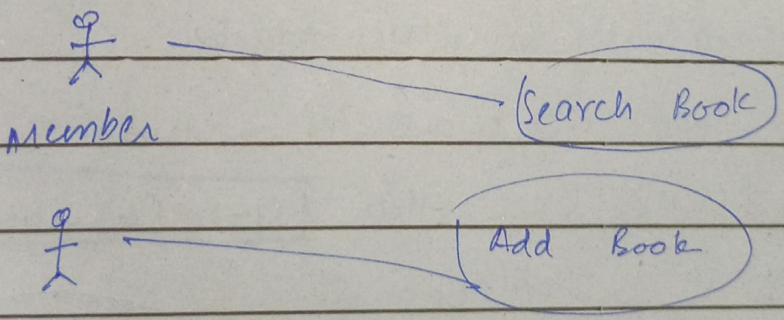
if cname variable is private

Date _____ Day _____

How to identify functions in class diagram:



- Book search function should be in Main. Because only main has list/array of books. GetTitle() well also be needed in Book.

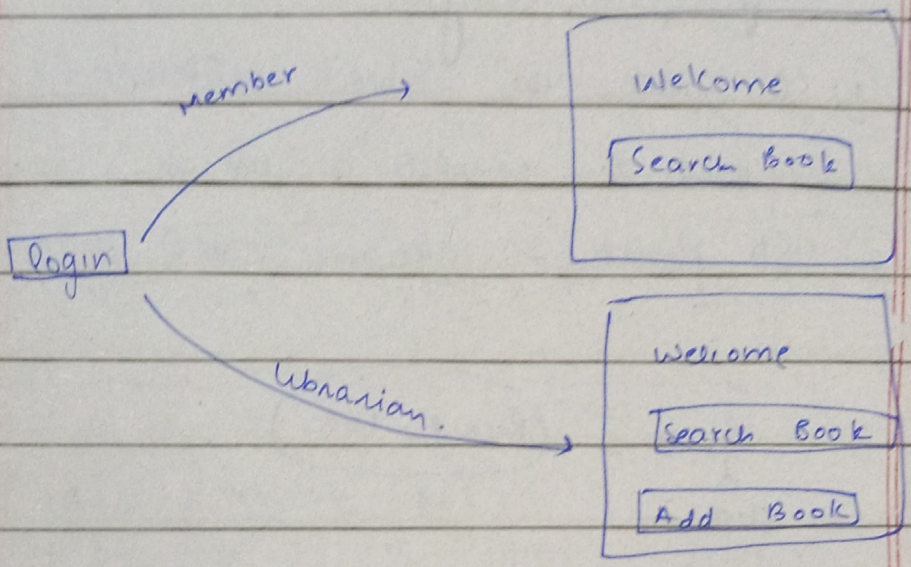


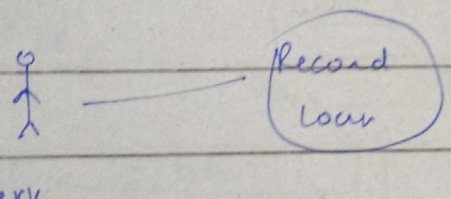
- Any user can access, construct of book and add a book. But we only want librarian to do so.

Date: Day:

- This can be done, by adding diff interfaces for each user.

Two interfaces






```
bool Record_Loan(Customer* c, Book* b) {
    Loan* temp = new Loan(c , b);
}

Loan(customer* c, Book* b)
{
    this->co.data = c.data
    this->bo.data = b.data.
    co.add(this);
    bo.add(this);
}
```

- Class Diagram → Static.
- Sequence // → Dynamic

Date _____ Day _____

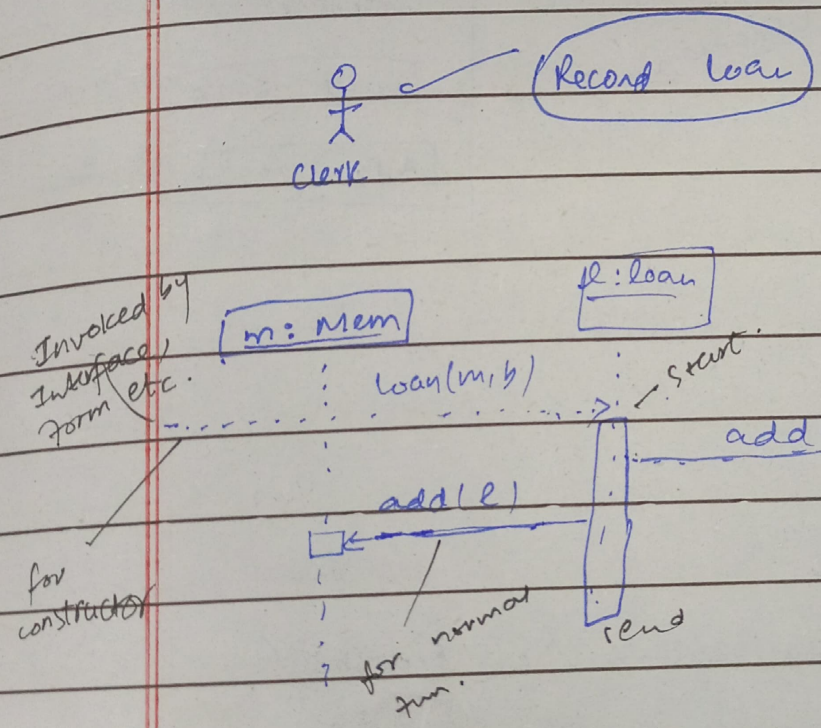
For both member, Book.

```
void add (loan *l) {
    this → L.append (l)
}
```

Sequence Diagrams :

(1) SD shows the object interactions required to complete a use case

(2) SD provides a dynamic view of system



(constructor)

small box → no function called in that fun

large box → more function are called in that fun. → add functions

Tail towards caller

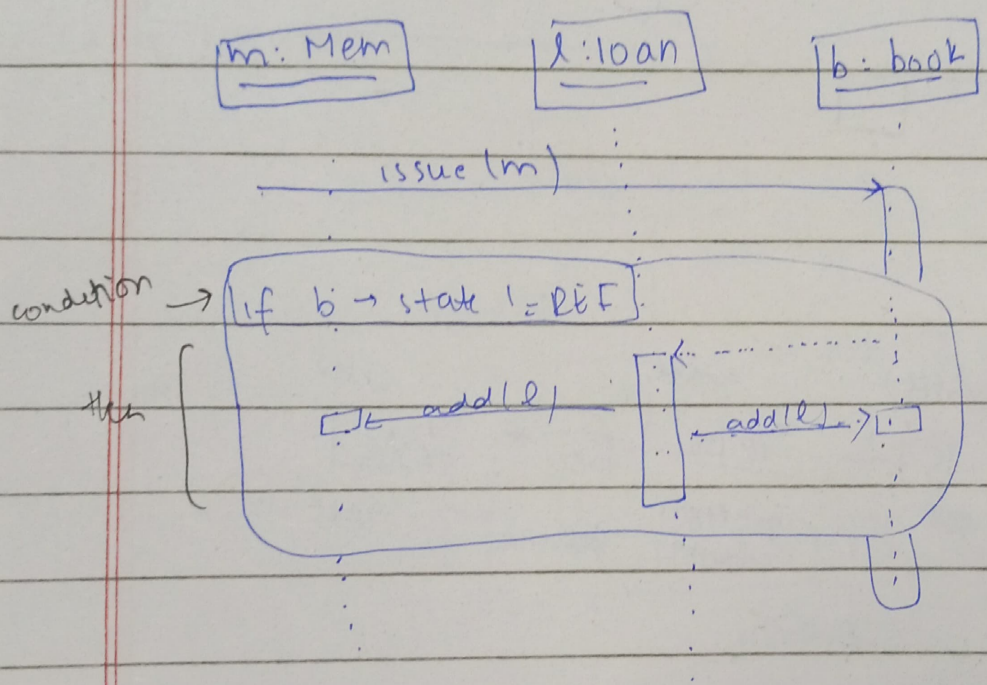
Head towards callee (which contains that fun)

Date: Day:

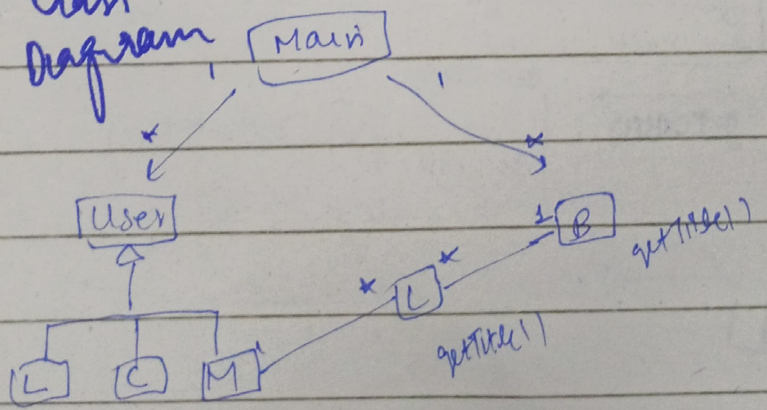
→ System can have more than one sequence diagrams for it. But has one class diagram.

if/else

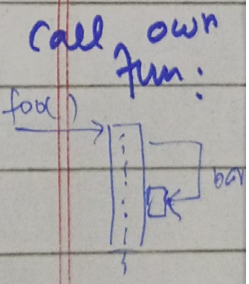
To show a loop* we build rectangle



Class Diagram

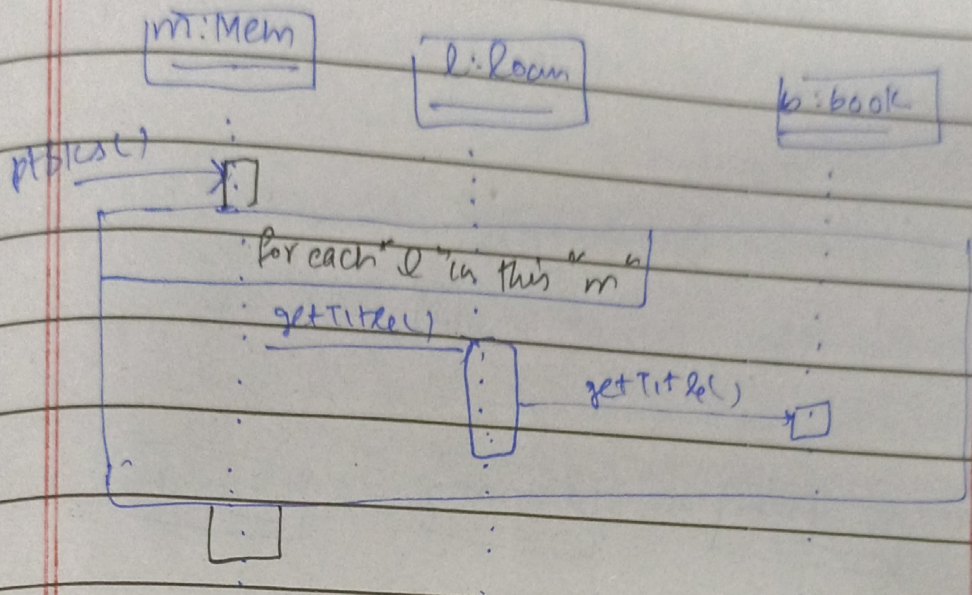


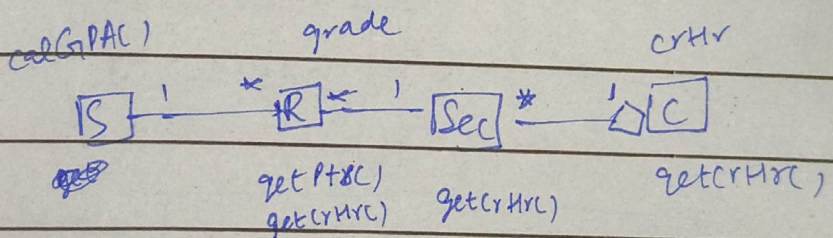
Call own fun:

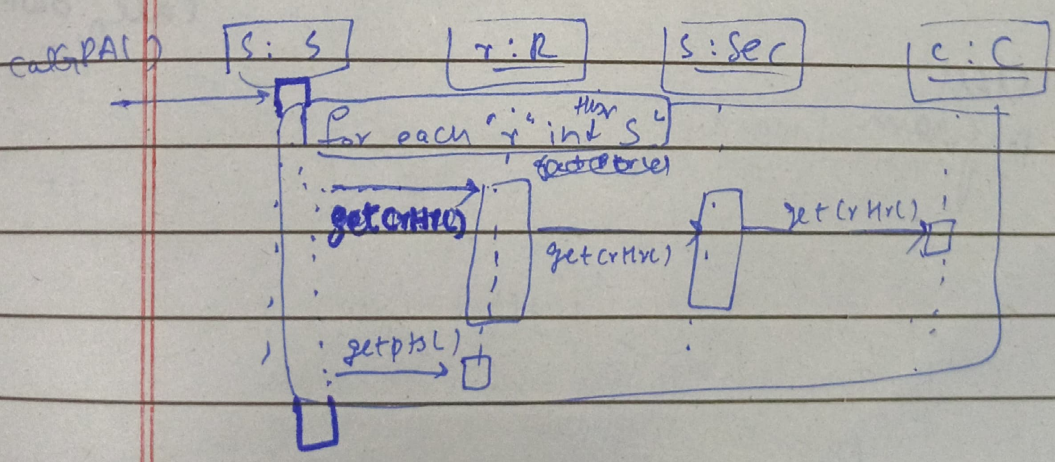


* Class diagram will be given. We will need to draw SD

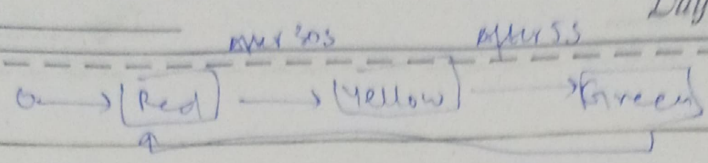
Show all books issued to a member :-





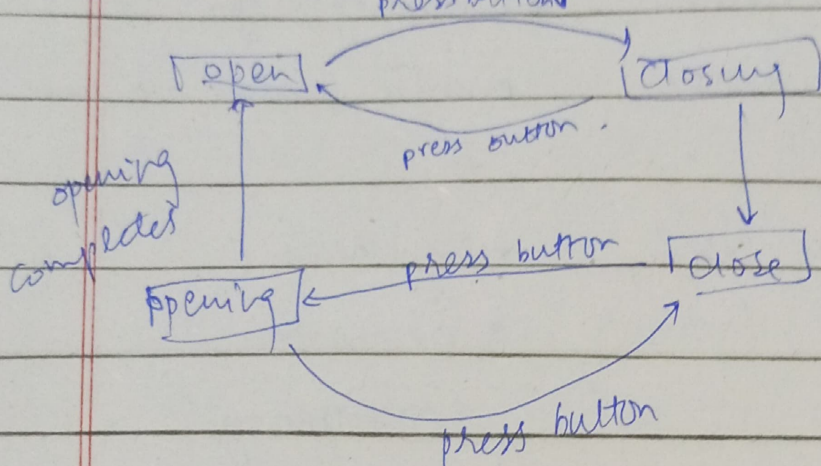


Date: Day:



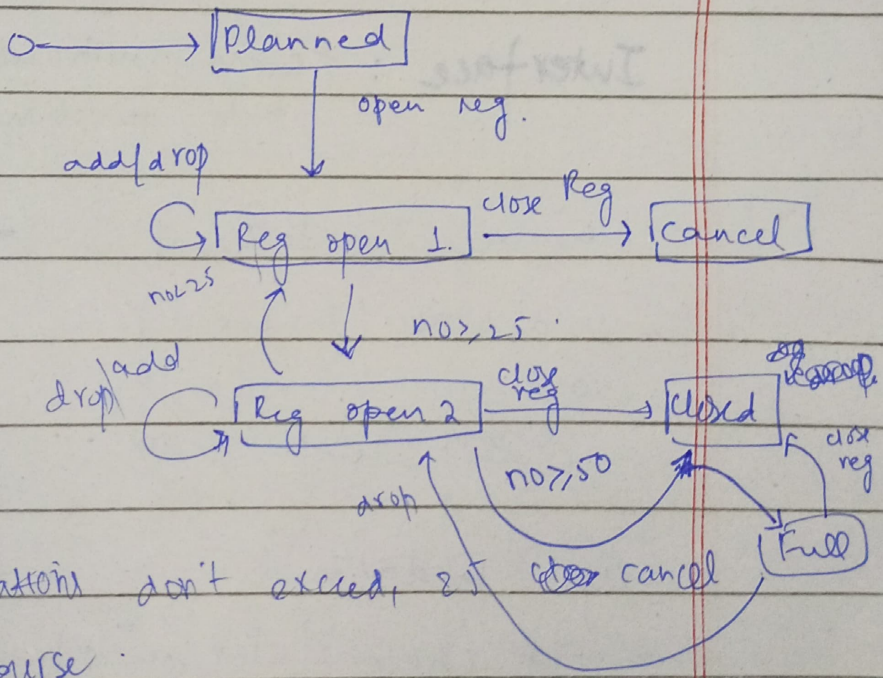
State Diagrams

Example:



Course Registration :-

Course:



- If registrations don't exceed, it cancel the course.

State Diagram :-

- State diagram shows an object that may have multiple states (and transition)
- Object changes its behaviour when its state changes.

Date _____ Day _____

State

- Object with multiple states, transitions (events)
- Arrows have label

Activity

- Flow of algo, business process, use case.
- No labels
- Decisions & concurrency

Interface : - class without data members only functions (pure virtual or virtual)

```
interface predator {          — Java
    void chase(Person P);
    void kill(Person P);
}
```

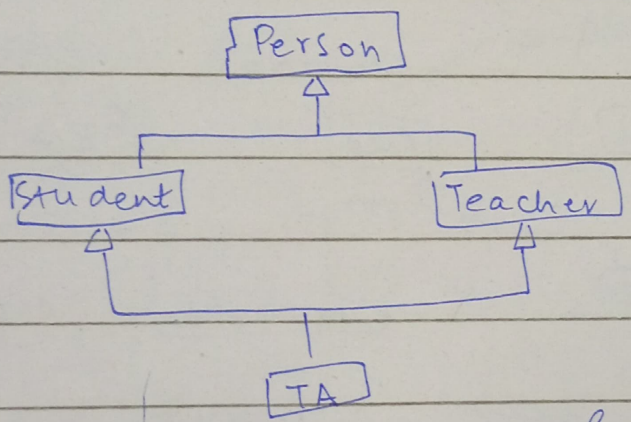
Both equivalent

```
class Predator {              — C++
    virtual void chase(Person* P) = 0;
    virtual void kill(Person* P) = 0;
};
```

• Pure virtual without body fun
• virtual with body of fun

Date: Day:

- while using interface keyword in Java, we can't create data member
- Don't need to put "=0" in Java.
- We can use getter & setters.
- only string in Java, no char.



Diamond problem

No multiple inherit ance in Java.

Student (name, roll): Person (name)

Teacher (name, rank) // //

Student (roll)

Teacher (rank)

Grad

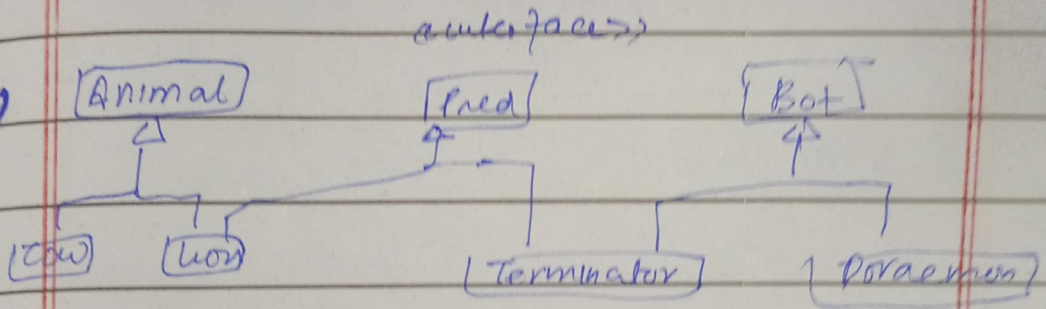
- TA(name, rn, rank, stipen): Person(name), Student(rn), Teacher

class student: public virtual Person {

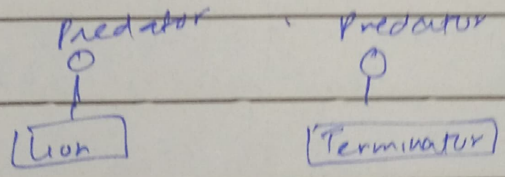
};

- With TA, new constructors of student & Teacher are called.
- with Student, Teacher old constructors are called

→ No two parents, of one class,
→ one can be interface, one class
interfaces can be multiple.

1) 

2) Another implementation



Cow & Doraemon are not predators,
Lion & Terminator are.

→ Lion, Terminator can be seen as predator

Date: Day:

```
void foo (Pred *a[], int n, Person *prey)
{
    for (i=0; i<n; i++)
        a[i] -> chase (prey);
}
```

lion, terminator will both start to chase prey.

```
class
interface pred {                    — template method
    void who ( ){
    cout << "Name of predator
      is " << getName( );
    }
    virtual char * getName ( )=0;
        string getName ( );
}
```

(template method → generic code and details later)

→ Interface calls fun of its children

```
Pred *p = new Lion ( );
  "   p1 =  "  Terminator ( );
```

(new Lion ( ) → fun getname( ) of lion will be called)

(Terminator ( ) → fun of terminator will be called in interface.)