Software Architecture:-

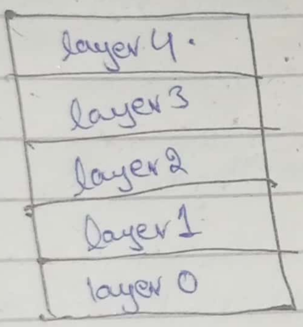
Software Structure.

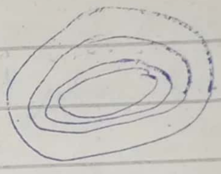
- Structure of a program comprised by its major constituent, their responsibilities & properties & Relationships & interaction b/w them

Architectural Design.

i.e. In a shopping mall there are sensors who senses any fire happening & after that they sprinkle water.

Layered style Architecture:(Major)





- partitioned into layers or groups.
- Use services of the layer below. & provide services to the above layers.

Uses & Invokes:-

Invokes:-

Module A invokes or call Module B (A triggers execution of B).

Uses:-

A uses B if B has correct version that must be present for A to execute correctly.

Scanned with CamScanner

Note that:-

Uses	Invoke.
✓	✗
✗	✓
✓	✓
✗	✗.

↳ Exist in System but has no relation with anyone.

Example.

$$\begin{tikzcd}
\text{Payment} \\
\text{Product} \arrow[u, "\text{uses.}"'] \\
\text{User} \arrow[u, "\text{invoke.}"'] \arrow[uu, bend left, "\text{Invokes}"]
\end{tikzcd}$$

layer Constraints:-
Static structure:-
layer are independent Not Use any layer but invoke.

Dynamic:-
Uper layer use only 1 below layer than its strict.
Upper layer uses more than 1 layer than it is relaxed.

Forming layers:-
1 level of abstraction:-
i.e Network communication layers.
2 Virtual Machine:-
i.e OS, interpreters.
3 Information hiding, decoupling etc.
i.e User Interface layers, virtual device layers-

layer Style Advantage.

Coupling. Modules depend on each other, slow processes,

Cohesion:-

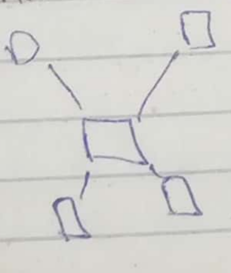
• Disadvantages:-
• Debugging , layer dependency $\overset{\text{order of}}{}$ on each other
•

Shared Data Style:-
• Data store at one place & Modules are connected to ~~each~~ that store.
• change in store may case change in every module.
• Server has high load.

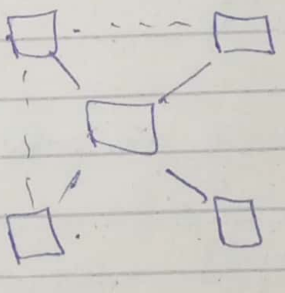
Definition:-

Two variants:-
• change in shared data ~~activo~~ activates accessors when the stores change Blackboard style:-
• functionality to client is Repository.
Server not control component but component use server:-

Thin client fat server.



fat client:- thin server:-



Advantages:-
Disadvantages

Event-Driven Style:-

- A System not having more functionality
- Sensors Detect & Event occur.
- Input / Out put

Implicit Invocation.
Not having a proper system that is run by any user.

Stylistic Variation:-
like traffic warden use's cameras who take pictures of every car.
- Synchronously, Asynchronously.
- Advantages
Easy to add & Remove components:-
Disadvantages:-
More Events less functionality

Model-View-Controller ~~Styst~~ style

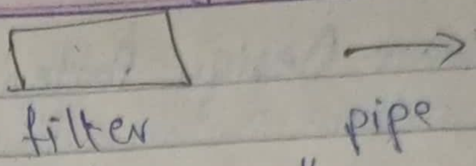
Pipe & filter Style.
- Amazon, fireware
Pipe : Execution

filter - change or pass from one to another

lexical Analyzer:
int x,y Means of anything.-
Syntax Error caught by it.

Acyclic Graph.

Semantic Analyzer:-



Car & vehicle

semantic meaning is same.

"Execute concurrently."

• linear Execution:-

Disadvantages:-

Error handling is Difficult.

Model View & Controller sytle.

Data

form in C#

that we can see

front End

we click botton & we went to the function. that function is the controller of the view.

Back End.

BL

Business logic

attached to the Data.

DL

Data logic

Data is independent, Used in View

Disadvantage.

MVC - Behavior:-

all the work that we done in SDA lab C#:-

• View & Controller are often hard to separate.

• Slow processing when connected to Database

Hybrid Architectures:-

Software Design Pattern.

A pattern is an outline of a reuseable Solution to a general problem encountered in a particular context

A pattern should have:-
context, problem, forces, Solution, Anti-Solution, related, patterns, references:-

i.e.
like static variable has 1 space in RAM & that is share or access acrross multiple classes to reduce space problem,

contex!:
Paragraph, Description of problem

Problem:-
Question type. short form of context with ?

forces:-
What kind of Constraints ~~do we have~~, which I have to take care of during the solution.

Solution:-
Solution is a class Diagram + Description.

Anti-Solution:- anti-Pattern
common Mistakes- made by programmers:-
Solve the problem & can't take care of the forces:- (also class diagram)

related pattern:- optional. related Solution:-.
Solution of a problem, ~~with~~ an alternate solution, solve the problem, ~~can~~ take care of the forces.
• Best one is Solution, other are related Pattern.

References:-
A pattern, who is the author of the pattern or in which year do they publish.

Imp:-
change in context also change the pattern.

• **Abstraction- occurance ~~procontext-~~ pattern-**

context-
• In a library, books have same title, Author, publisher etc. but bar-code is different

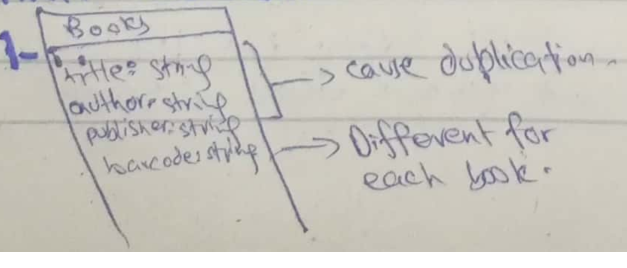
• A flight has same destination, leaving station but has different date & passengers.

Problem:-
how can we represent such set of occurances in class diagram.

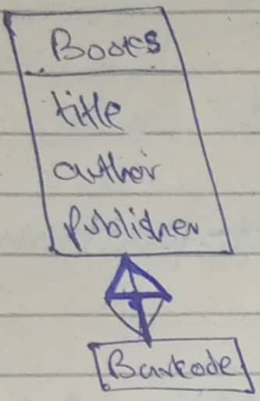
Forces:-
Duplication & Flexibility.

• less efficiency of program
• take care of problem in such a way that common data should not be in multiple classes.

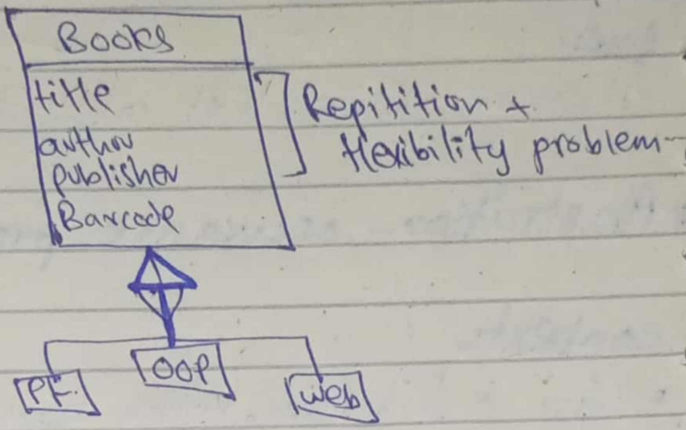
• easily add & remove occurance.
• previous one will not be disturbed

Anti-Solution:-
1-


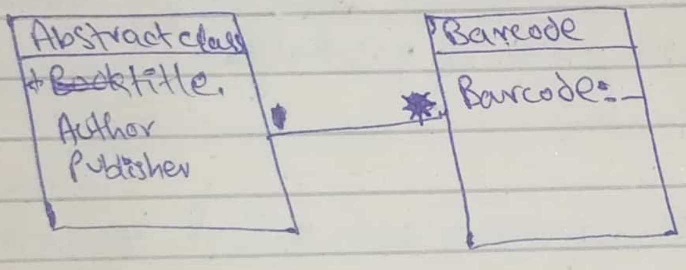
2:- Common information is in parent class & different is in inherited class:-



3:-

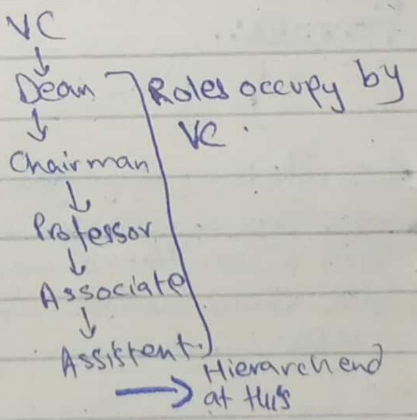


Solution:-



★ General Hierarchy Pattern:-

Context:-



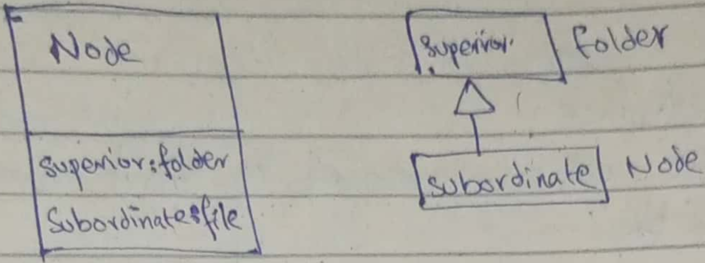
Problem:-

How can we show hierarchy in our class Diagram in which some objects don't have subordinates?

Forces:-

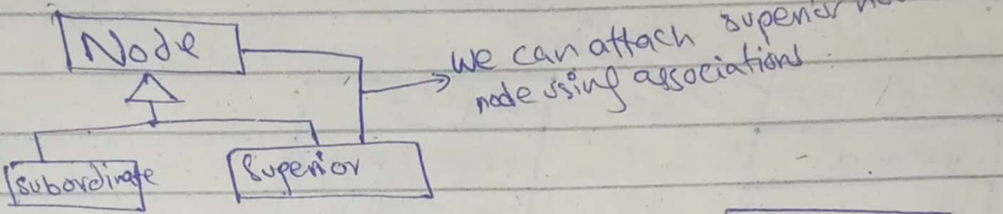
- Flexibility → make multiple folders into the parent folder can't act any effect on parent.
- Common features:-

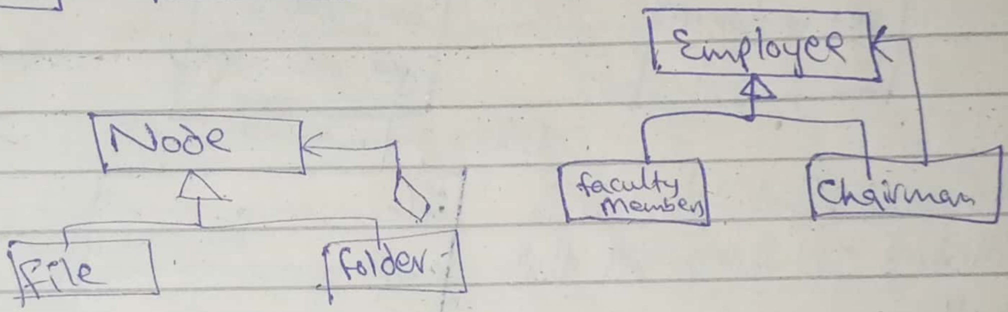
Anti-Solution:-



one folder has multiple folders
No Flexibility

Solution





3) Singleton Pattern:-

Context:-

University & company name is same for all student or Employees (university/company name is saved in class)

Pattern:-

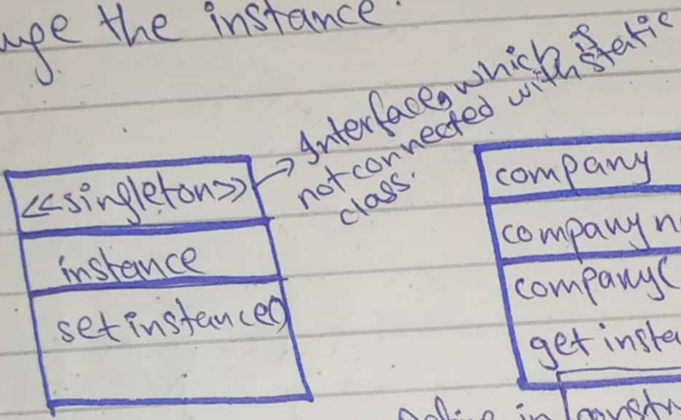
How do you make sure it is never possible to make more than one instance of a class:-

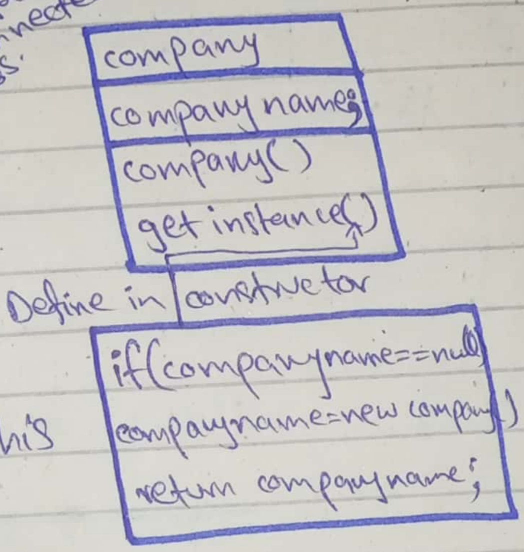
Forces:-

Singleton instance must be public & should not be changed (Assecible to all classes)

Solution:-

→ private class: instance.
→ public class static which creates & store the instance & return it
→ ~~Pu~~ Private constructor: Ensures that no other class change the instance.





```
if(companyname==null)
companyname=new company()
return companyname;
```

No anti-Solution & Related Solution of this pattern:-
static keyword make a class at the backend also that stores data in variable.

4- Player Role Pattern:-

Context:-
A student can change his role from full time to Part time or from undergraduate to Post-graduate.

Problem:-
How do you best presents players & roles

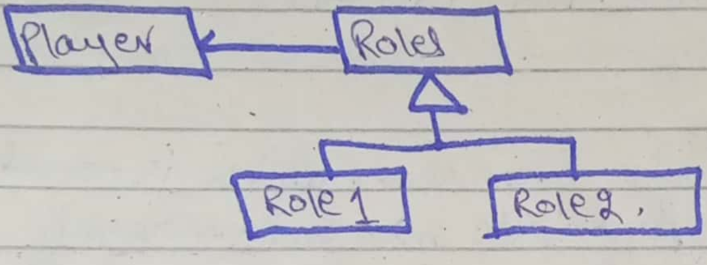
where player can change roles?

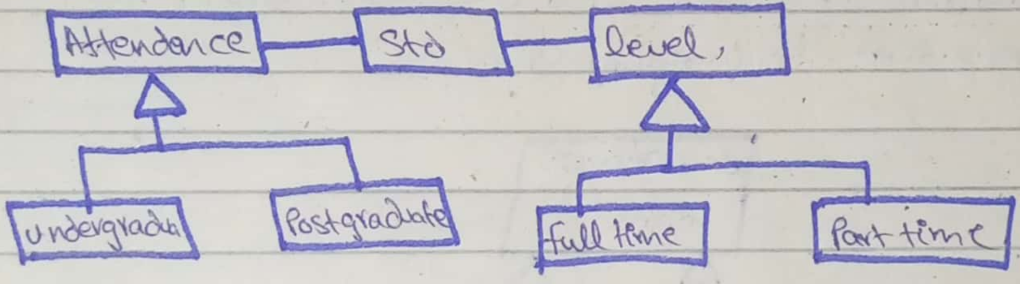
Forces:-

[illegible] [illegible]
Encapsulation & multiple inheritance

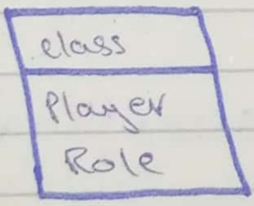
Solution:-

→ Make a role class:-
→ All roles will be subclasses of this role class.-
→ Attach roles class to player class with simple association:.

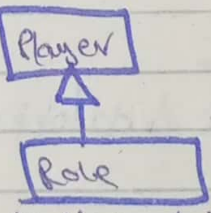




Anti-Solution:.

1. 

Encapsulation is neglected

2.- 

Multiple Inheritence occurred.-

5:- Proxy Pattern:-

Context:- you have huge Database & working with heavy weight classes which access this data but you cannot bring all data in main memory. And also you don't need whole data to work on

Problem:-

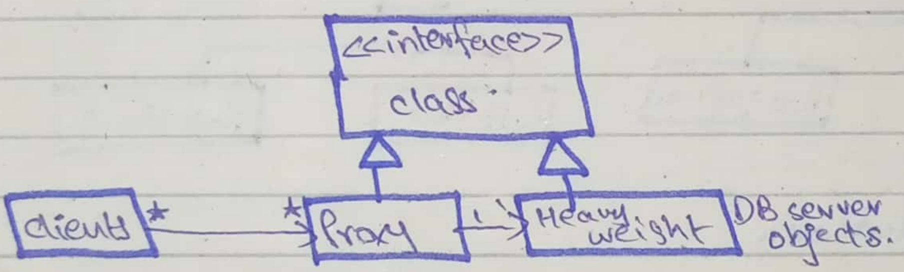
How do you reduce the need to load heavy weight object/data from DB/server

Forces:-

load only objects/data from server which is needed:-

→ Don't bring all objecte into main memory:.

→ How data is stored and loaded it should be transparent to programmers:-



Anti-Pattern:-

load whole data/objects from DB/server to main memory:-

6:Delegation Pattern:-

Context:-

You are working in a class & realize text another class already have implementations of the function, you are creating in first class.

Problem:-

How do you efficiently use functions that already exists in other class:-

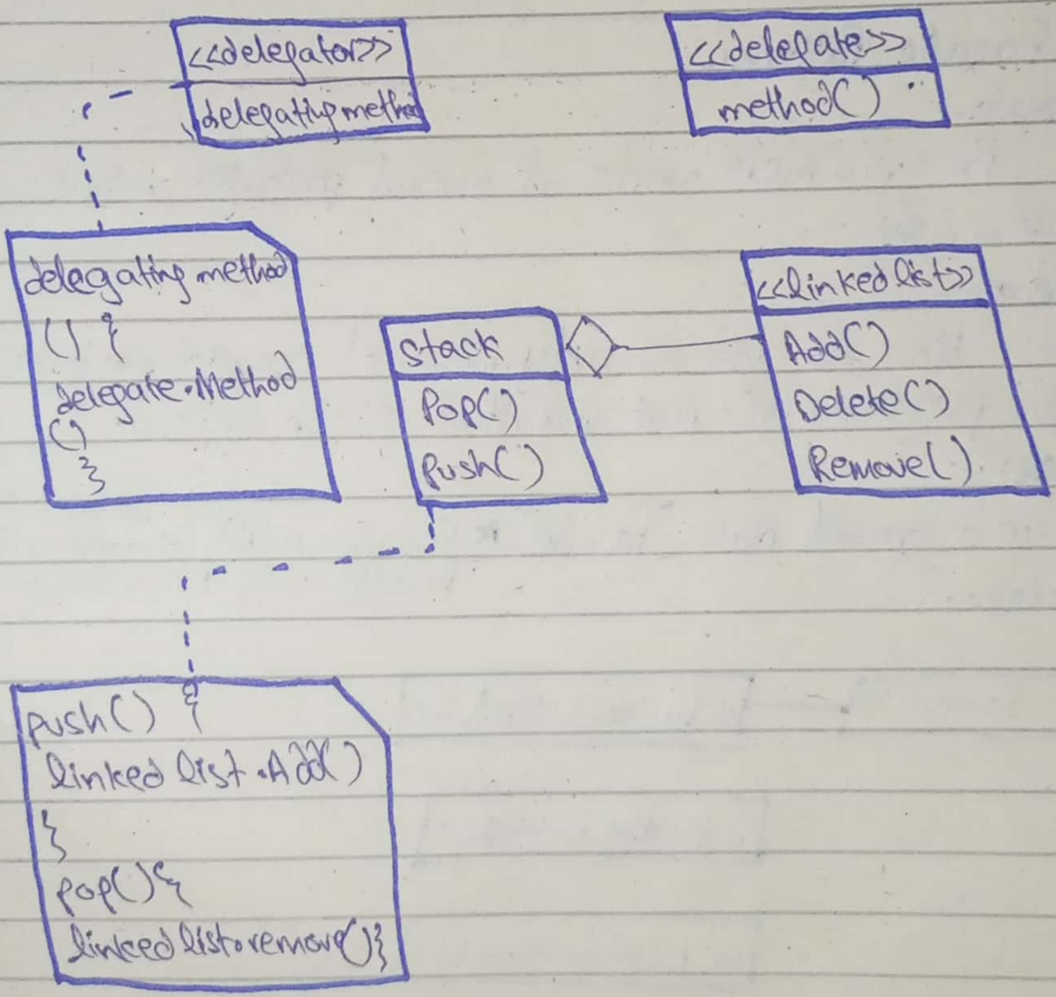
Forces:-

Minimize development, cost, complexity:

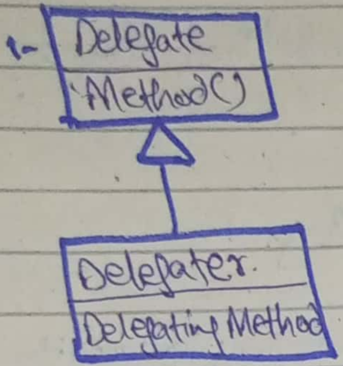
Solution:-

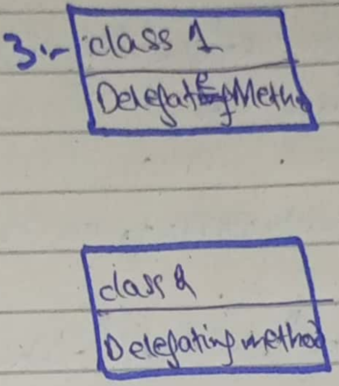
A class called "Delegate" already have implementation of a function:-

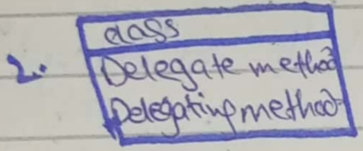
→ create class "Delegator" & connect this to "Delegate" using Association:-



Anti-Solution:-

1-


3.-


2.


7.- Facade Pattern:-

Context..

Programmers works of several packages while programming.

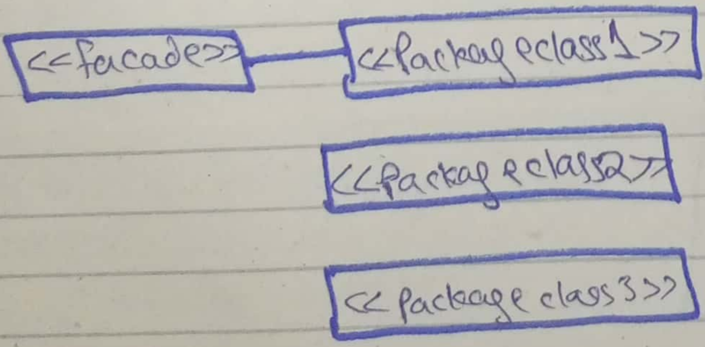
Problem:-

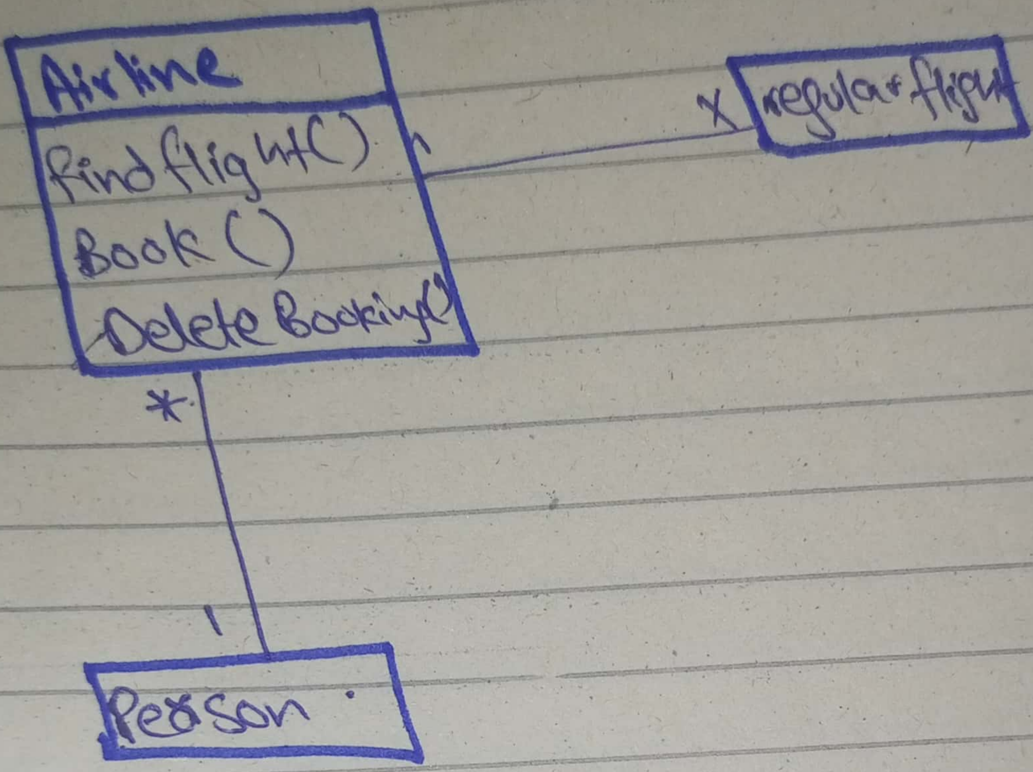
How do you simplify view of complex packages for programmers: must understood Entire system.

Forces:.

* Create a special class "facade" * facade will contain all ~~nected~~ with Package using Association.

Solution..





Scanned with CamScanner