__/__/__

Object Oriented Programming

*) Creational patterns : different ways to create objects
→ these ways increase flexibilities and allow reuse of existing code. the creation logic is decoupled from the implementing system
*) the class which implements the object is different from the class which creates the object

*) Structural patterns : how to assemble classes and objects into larger structures while keeping these structures flexible and efficient, how objects relate to each other, tell us about relationships between objects

*) Behavioural patterns : communication between the objects and assignment of responsibilities to the object.

SOLID principles

1) single responsibility principle (SRP)
2) open / closed principle
3) Liskov's substitution principle (LSP)
4) Interface segregation principle (ISP)
5) Dependency inversion principle (DIP)

*) good code explains itself, and is extensible to modification (bad code is confusing and breaks on modification, [Rigidity : Dependency of modules, to change something you must change all dependent modules]
→ coupled systems make code rigid and make it hard for isolated change to the code (Rigid Code)

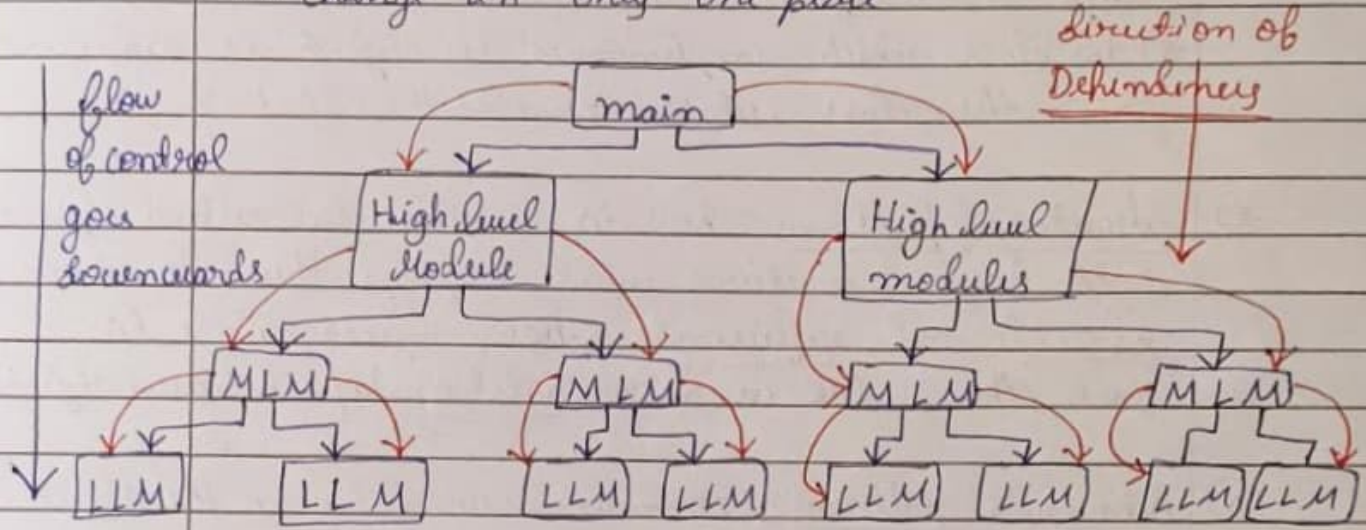
→ Fragile code : breaks in strange ways which we can't predict, completely un-related code breaks because of some arbitrary change in the code base

fragile code

we want to reduce unwanted undesirable dependencies or coupling, we want our code to be as decoupled as possible

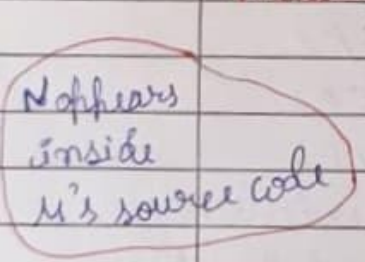
Ripple Effect

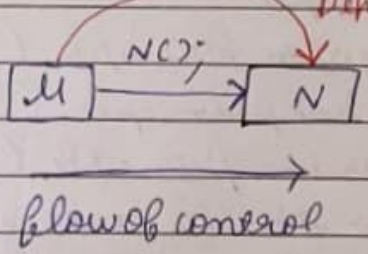
rigidity: makes it hard to make isolated changes
fragility: tendency of the code to break at many places, when occurs when you are making the change in only one place



MLM: middle level modules, LLM: low level modules
HLM high level modules

*) the higher level modules know about the lower level modules





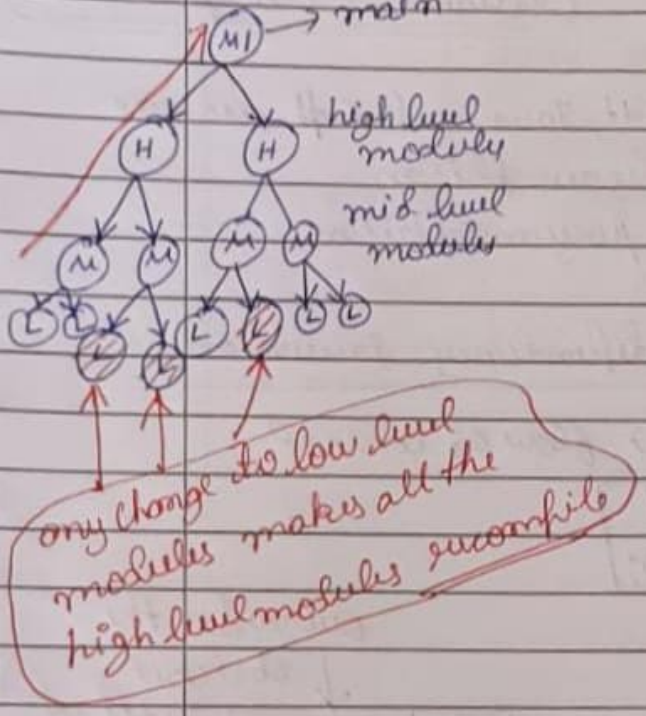
*) N appears inside M's source code
*) thus M knows about N

*) flow of control and dependency point in the same direction, if there is any change in N, M has to recompile (M also changes)
*) M is dependent on N

*) in the diagram mentioned above the higher level modules know about lower level modules

this violates Dependency inversion principle

*) High level policy is polluted with low level details, we can't read this code easily __/__/__
→ hard to grasp high level Idea, when the code is filled with low level complexities



*) change to low level modules affects the high level policy
*) this leads to rigidity
→ hard to make changes at Low Level modules as it propogats upwards and makes changes to high level modules
Because of tight coupling between HLM and LLM
high level and low level modules

OO design characteristics ① Encapsulation ② inheritence ③ Polymorphism

C had perfect encapsulation: we can forward declare or pre declare our functions and data structures in a header file, *) include the header file in a C-file and implement them in the C-file

→ users included the header file and could'nt see the implimentation (data values were not visible only function signatures were exposed and only names of the data structures and not their members were exposed)

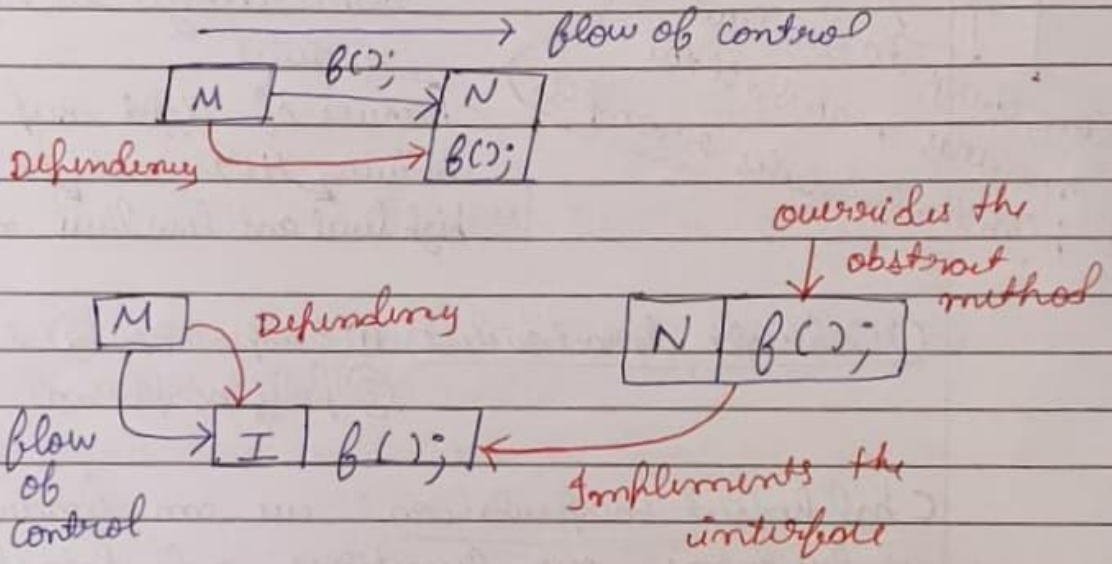
C++ came up with OOP and variables ~~which~~ were visible, it fixed this using the Public, Private and Protected key words

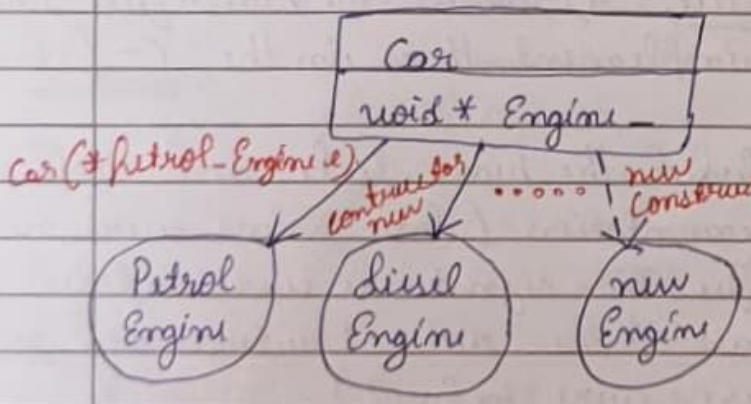
__/__/____

thus OOP ~~languages~~ languages these days has Partial Encapsulation (weakened Encapsulation)

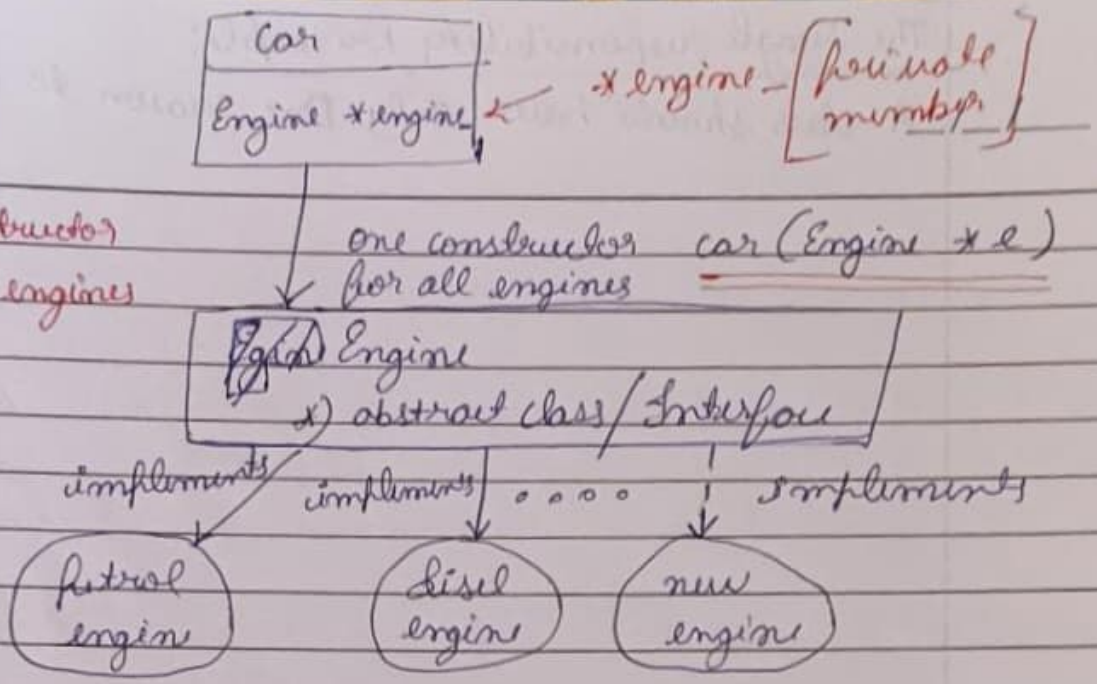
*) In OOP languages like C++, Java and C# we use inheritance to get Polymorphism (example: run time Polymorphism

How Polymorphism fixed Dependency inversion





*) when we make changes / create a new engine we need to create a new constructor that allows the car to have that engine



Interfaces allows us to invert any Dependency we get control over the dependency structure there by we can write non-rigid, non fragile and reusable code.

*) OO design is about managing dependencies by selectively inverting dependencies, and ensure rigidity, fragility and non-reusability