So LID

- S → Single Responsibility Principle [SRP]
- O → Open for Extension Close for Modification or Open Close Principle [OCR]
- L → Liskov Substitution Principle [LSP]
- I → Interface Segregation Principle [ISP]
- D → Dependency Inversion Principle

SRP → A class should have only 1 reason to change
Like in Spring Boot we have Controller to map to URL, Service class to have all Business logic & DAO class to have DB operations so All classes have Single Responsibility

OCR → Suppose DAO class save to DB Now new requirement comes save to FileSystem. But our DAO class is Already Live
If we add Add to FileSystem to DAO it can create Bugs
Soln is Create a Interface DAO
we have its implementation DbDAO, FileDAO One saves to DB & other to File so If something new comes we can store it in a new implementation class so Open to Extension & close to Modification

LSP → Class B is subclass of class A, then class B should be able to replace class A without breaking the program
child should Extend capability of parent not narrow it down [Like Real life] A good son is one who Ise property, prosperity than Parent of family

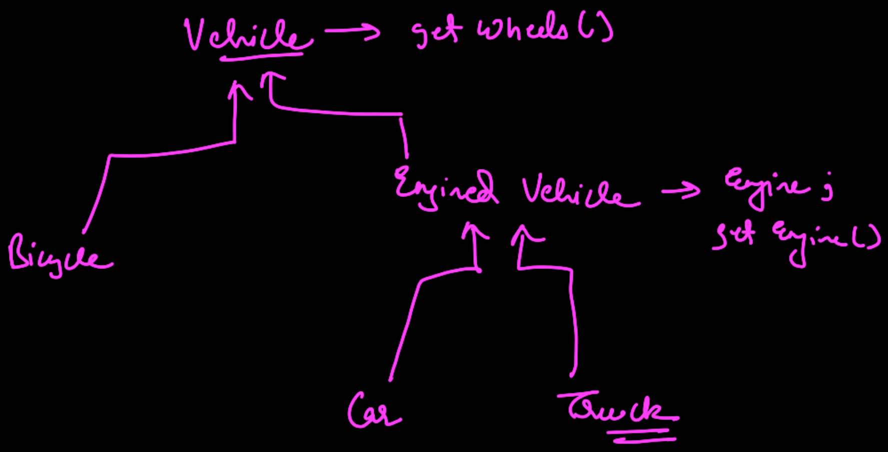
ISP → Interfaces should be such that client should not implement unnecessary function

DIP → Class should depend on Interface rather than implementations

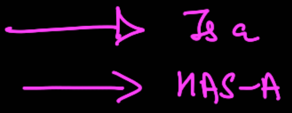
LSP we know the problem we need to substitute parent by child without breaking code

Problem occurs when child class downsize capability of parent like Parent was a Transport class having Engine but child is Bicycle class having No Engine so throwing Exception in get Engine().

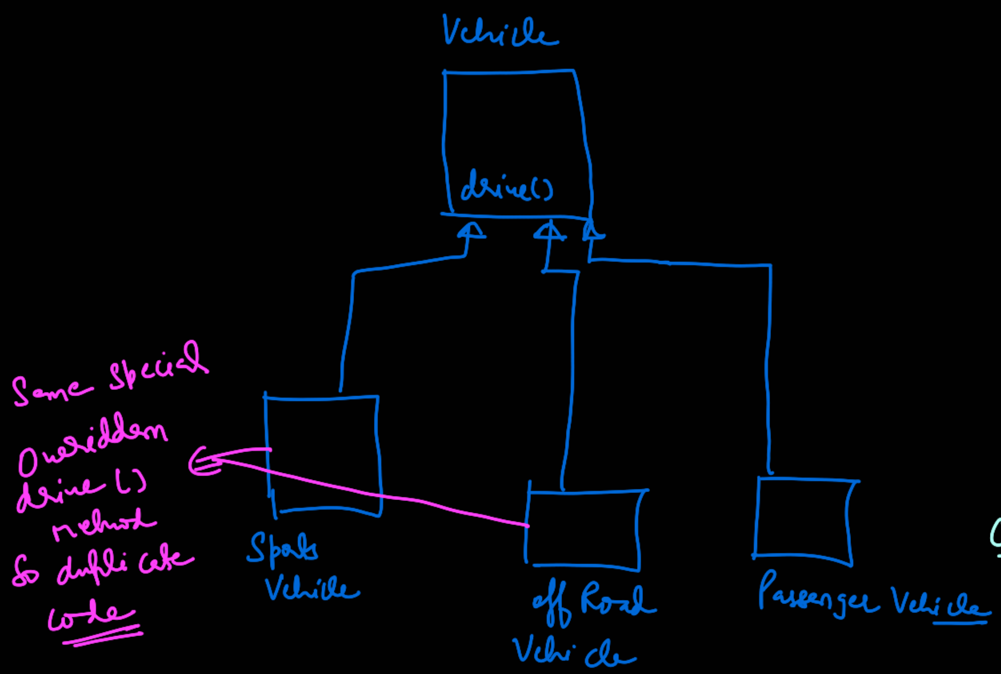
Sol^n →



So here in Parent class we have only Generic method like All vehicle has wheels !! So Vehicle dont have Engine now so Bicycle dont downsize capability of Vehicle



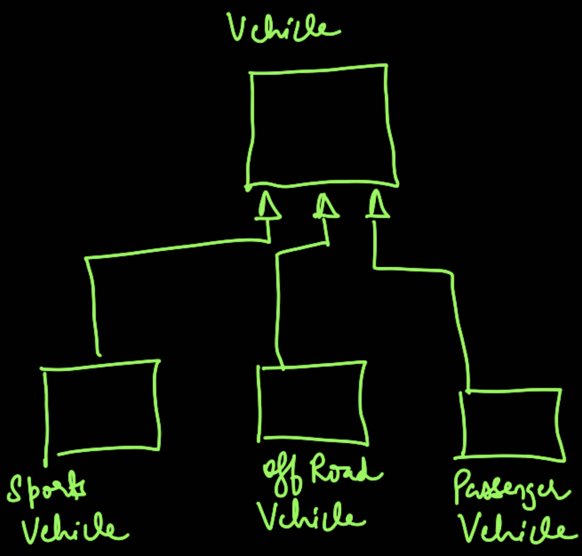
Lets see strategy Design Pattern

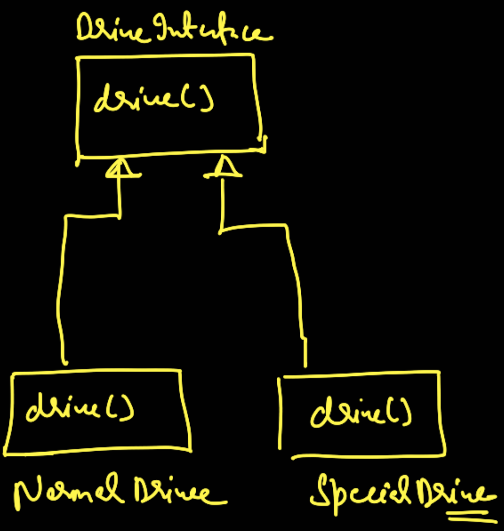


There can be duplicacy in child class as Many child class can have same method body
Siblings having duplicate code is problem

Now As child grows duplicacy grows

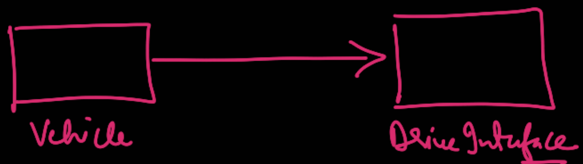
Solution by Strategy Pattern





for Drive logic create a seperate Interface & implement that

New Vehicle has-A Drive strategy



or Name it as Drive Strategy

```
class Vehicle {
    Drive Strategy drivest;   ] Now which Drive strategy is put
                                is decided by children
    Vehicle (DriveStrategy driveSt) {
        this.drivest = drivest;
    }
}
```

Dont do new DriveStrategy() as It will not be scalable Instead inject by a constructor

Strategy is Just Constructor Injection

[ye toh bas constructor Injection hi hai]

used when sibling sharing same code which is not in parent class

Code in 014 of ineuron