DATE :__/__/__
PAGE :_____

Solid Principles of OOPs

S – Single Responsibility principle
O – open / closed principle
L – Liskov Substitution principle
I – Interface Segmented principle
D – Dependency Inversion principle

Advantages of the following principles

Help us to write better code.
- Avoid duplicate Code
- Easy to maintain
- Easy to understand
- Flexible Software
- Reduce Complexity

1. Single Responsibility

"A class should have only one Reason to change".

This means that every class should have single responsibility or single job or single purpose.

DATE : __/__/__
PAGE : _____

Ex

```
class Marker
{
    String name;
    string color;
    int year;
    int price;

};


class Invoice
{
  Private Marker obj;
  Private int quantity;

  Public Invoice (Marker obj, int quantity)
  {
        this // constructor
  }
  Public int CalculateTotal()
    {
        int Price = (marker.obj.price * this.quantity)
        return price;
    }
  Public void printVoice()
  {  // Print the voice.
  }
  Public void saveInvoice()
    {  // Save data into DB
    }
};
```

class Invoice — calculates total
— Print Invoice
— save data to DB.

If calculation logic changes → Invoice change
If printing logic changes → "
If DB logic changes → "

This class is not following Single Responsibility Principle.

```
class Invoice {

 Private Marker obj;

 Public int calculate()   // This class calculates
 {   // calculate total      the total.
 }
};

class InvoiceDao   // This class saves
{                  the invoice to DB
   Invoice obj;
};

class InvoicePrinter   // This class print
{ private Invoice obj;    the invoice

};
```

2) open/closed Principle

This principle states that "Software entities (classes, modules, functions etc) should be open for extension, but closed for modifications which means you should be able to extend a class behaviour, without modifying it

Ex:
```
class InvoiceDao
{
    Public void SaveToDB()        // This class save Invoice to DB
    {
    }
};
```
↓
This class is tested, Live and has traffic.

Now new request of saving to file comes:-

```
class InvoiceDao
{
    Public void SaveToDB()
    {
    }
    Public void SaveToFile()      // This does not follow open/closed principle
    {
    }
};
```
↓
As we modified already existed, Live, Tested class.

```
Interface InvoiceDAO
{  Public void Save(Invoice ob);
};
```

Solution:

```
class InvoiceDao DB implement InnvoiceDao
{
    Public void SaveToDB()
    {
    }
};

class FileInvoice implements InvoiceDao
{
    Public void SaveToFile()
    {
    }
};
```

3. Liskov's Substitution Principle

If class B is subtype of class A, then we should be able to replace object A with B without breaking the behaviour of Program.

OR

child or Derived classes must be substitutable for their base or parent class.

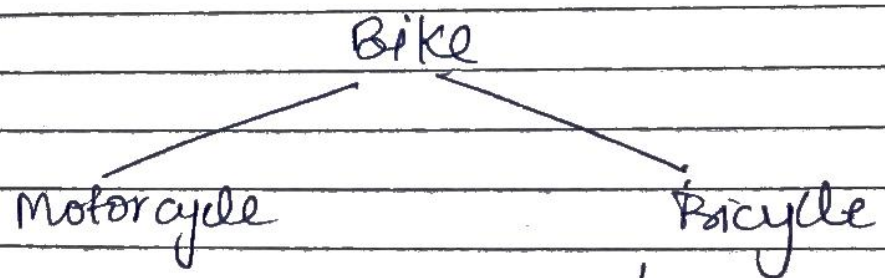
Ex-

```
Interface Bike{ }
    void turnONEngine();
    void accelerate();
}
```

DATE : __/__/____
PAGE : _____

```
class Motorcycle implements Bike
{
    boolean isEngineOn;
    int speed;

    Public void turnonEngine()
    {
    }
    Public void Accelerate()
    {
    }
}

class Bicycle implements Bike
{
    Public void turnOnEngine()
    {
    }
    Public void accelerate()
    {
    }
}
```

// Since Bicycle does not has engine
↓
This will disrupt the program nature!



↓
This reduces the capabilities of Parent class by not implementing

DATE : __/__/____
PAGE : _____

4. Interface Segregation Principle.

It states that 'Do not force any client to implement an interface which is relevant to them.

Ex. Interface RestaurantEmployee {

```
    void washDishes();
    void serveCustomers();
    void cookFood();
}

class waiter implement RestaurantEmployee
{
    Public void washDishes()
    {                                  // Not the work of
    }                                     waiter
    Public void serveCustomer()
    {
    }
    Public void cookFood()  // Not the work of
    {                          waiter.
    }
}
```

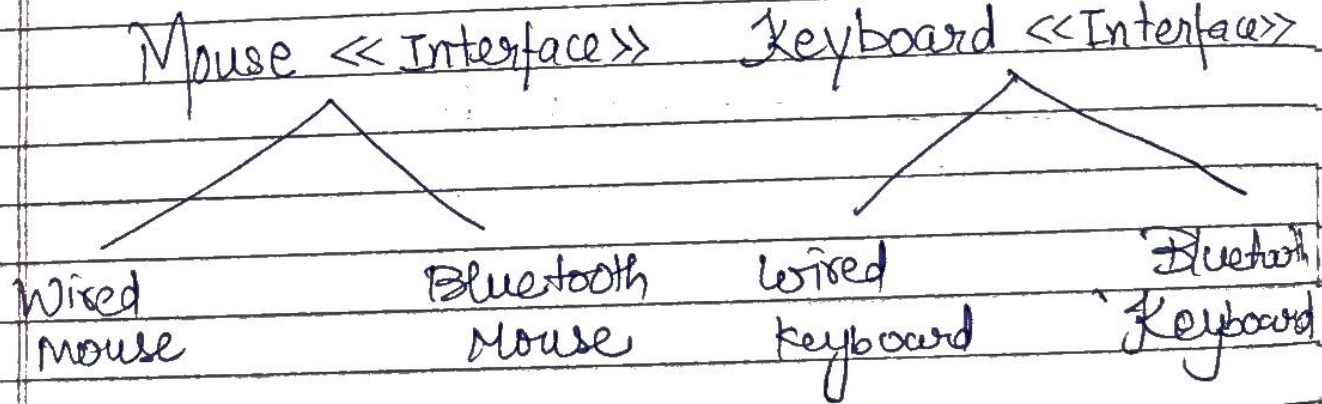
∴ Solution :- we will break the interfaces into small pieaces.

```
interface WaiterInterface{
    void ServeCustomer();
    void takeOrder();
}

interface ChefInterface
{
    void cookFood();
    void decideMenu();
}
```

5) Dependency Inversion Principle

It states that "high level modules should not depend on low-level modules. Both should depend on Abstraction.

class should depends on interface rather than concrete classes.



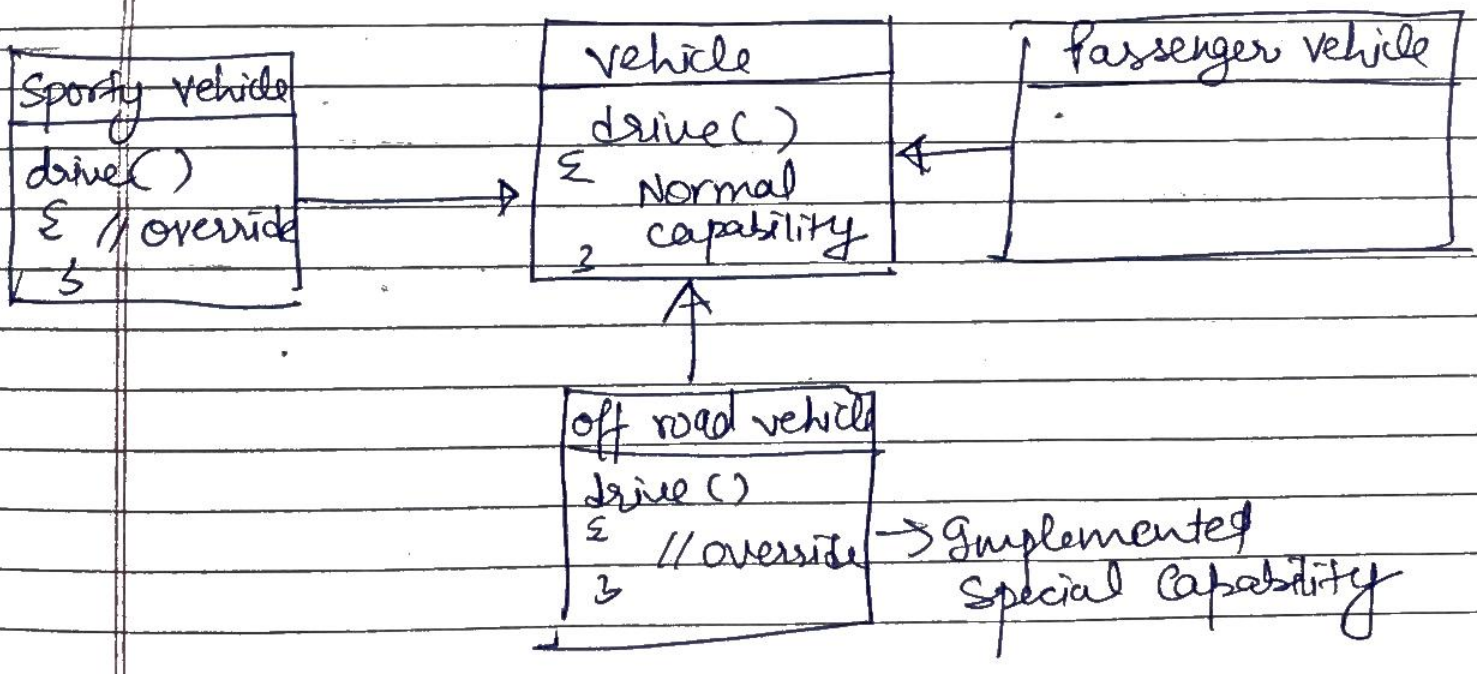
DATE : __/__/__
PAGE : _____

```
class MacBook {

Final WiredKeyboard keyboard;
Final WiredMouse mouse;

Public MacBook()
{
    keyboard = new WiredKeyboard();
    mouse = new WiredMouse();
}
```

Design Pattern → are the principle for the oop.

1. Strategy Design Pattern

is-a → is has a

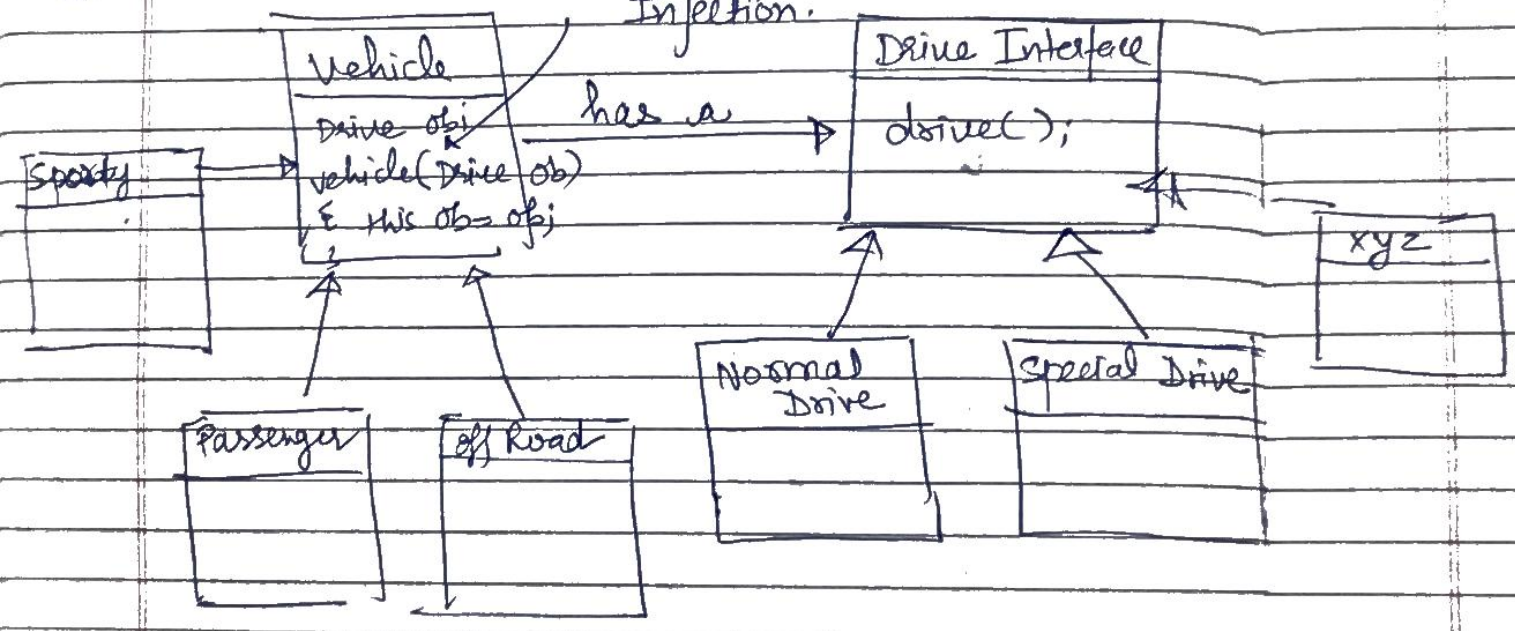


Problem Arises when two derived classes have same capabilities, this increases code redundancy.

Instead of using parent or base class capability the derived classes use or has common or capabilities then problem arises like
↳ No code reusability
↳ No Scalability

As we ↑ child, duplicacy in code ↑

Solution:-



Now the Base classes ie sporty, passenger, off road classes will decide which drive ~~class~~ Interface strategy they wants to implement. as Base class is not implementing a particular Drive ie (Normal, special, xyz).