Agenda

- LSP
- IS
- DI

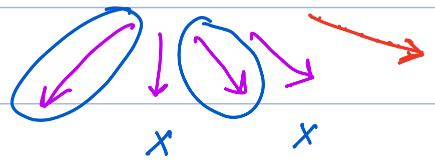
V0

Bird

fly()
makeSound()

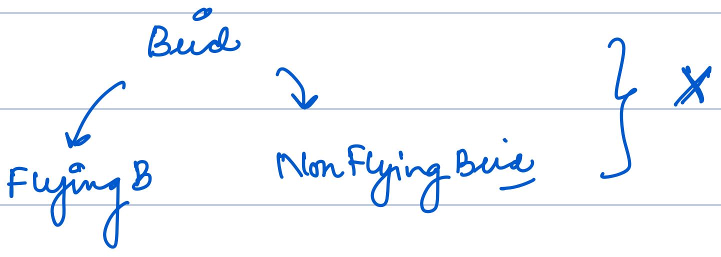
V1

abstract Bird

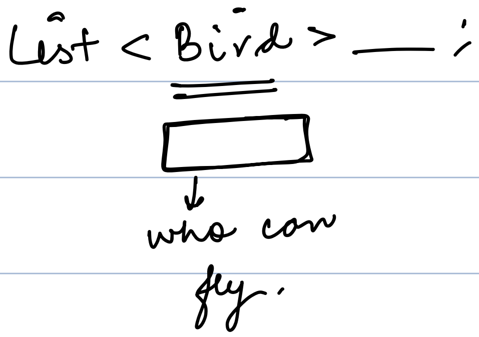


V2

Some Birds can fly
some can't.

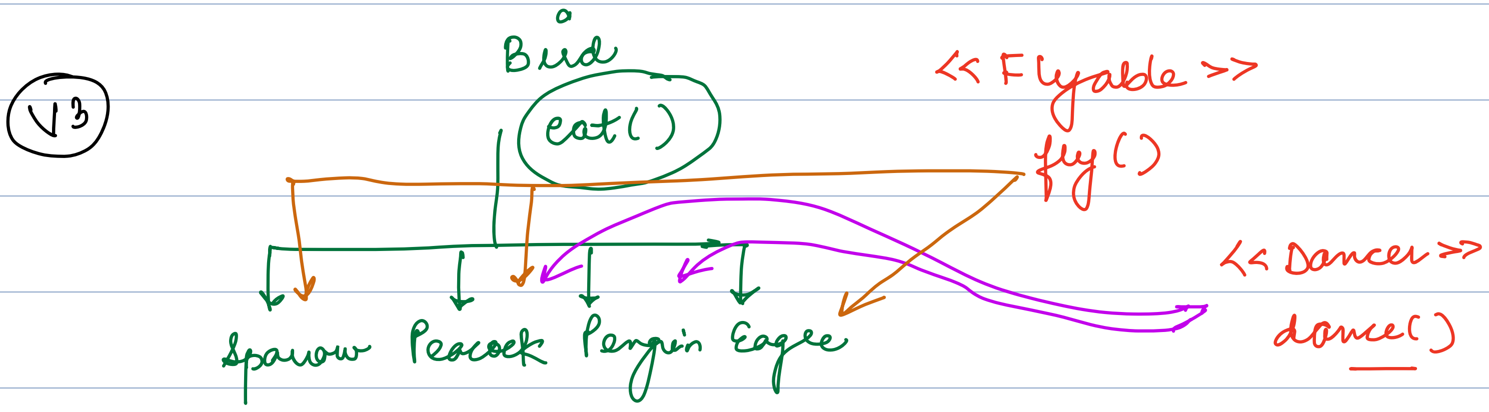


Problem statement : Some Birds demonstrate a behaviour & some doesn't.



classes : — entity

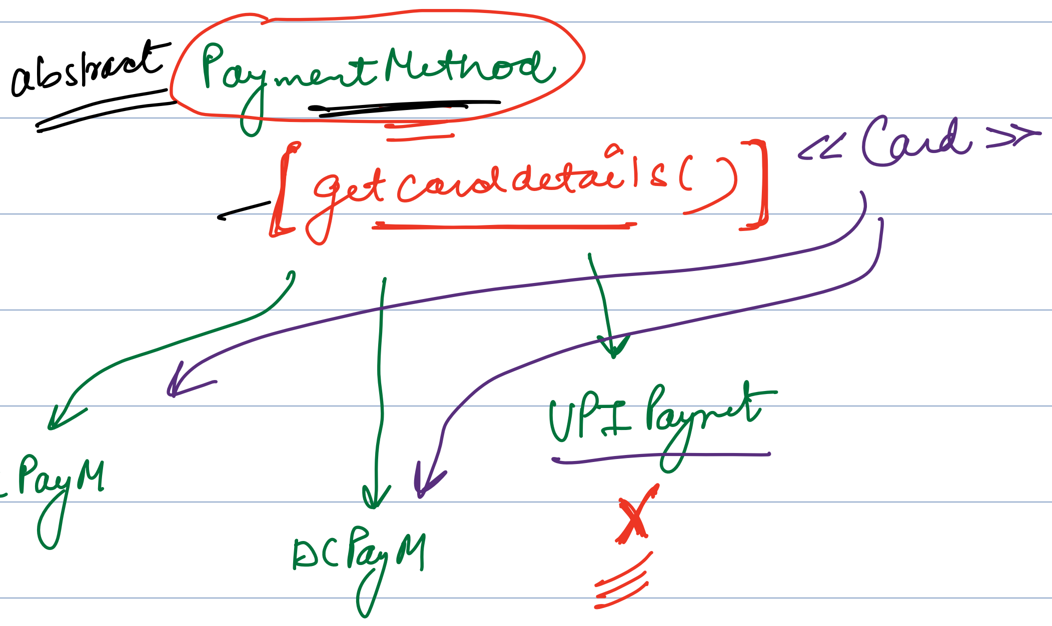
Interfaces ← behaviour

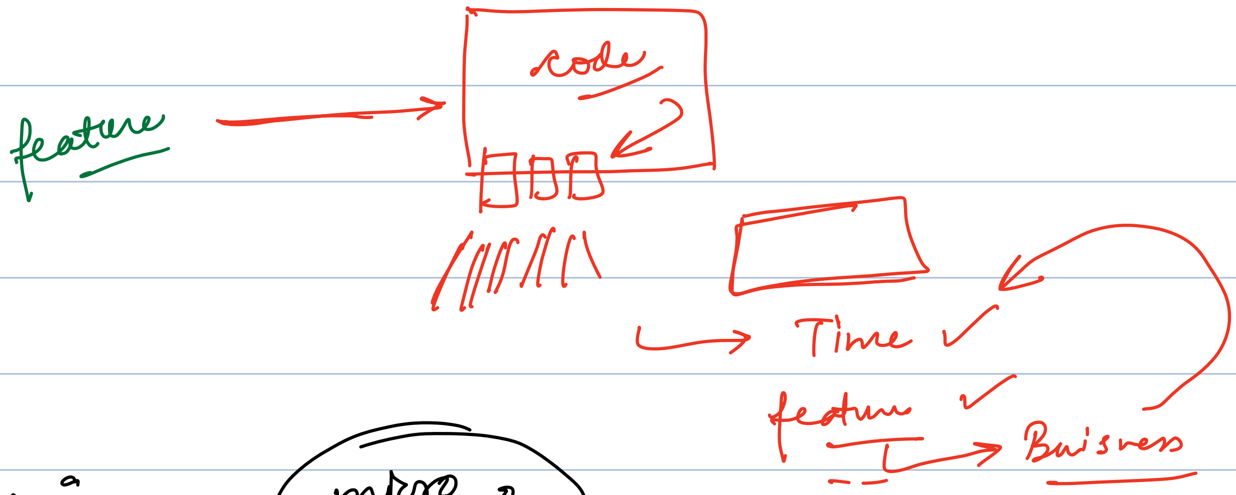


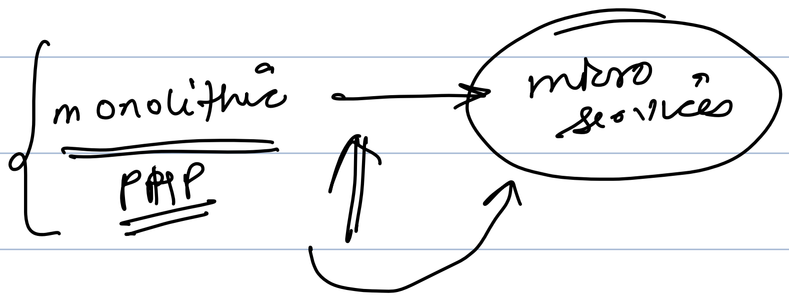
Liskov's substitution Principle

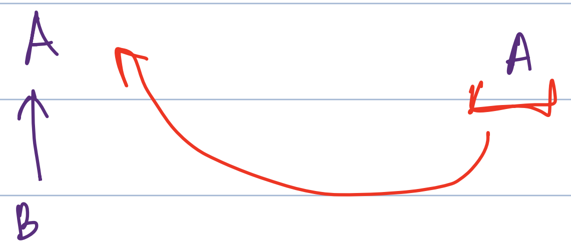
Object of an sub-class should be as-IS substitutable in the parent class ref without any code change.

```
Bird b = new Penguin();
b.fly()
```

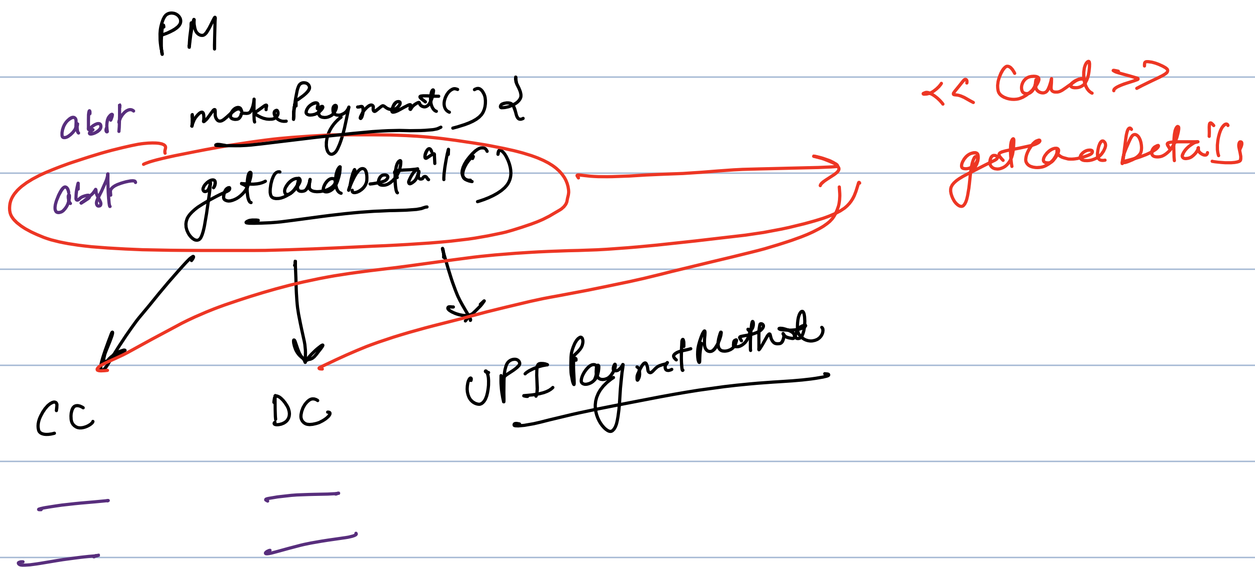








```
A x = new B();
```

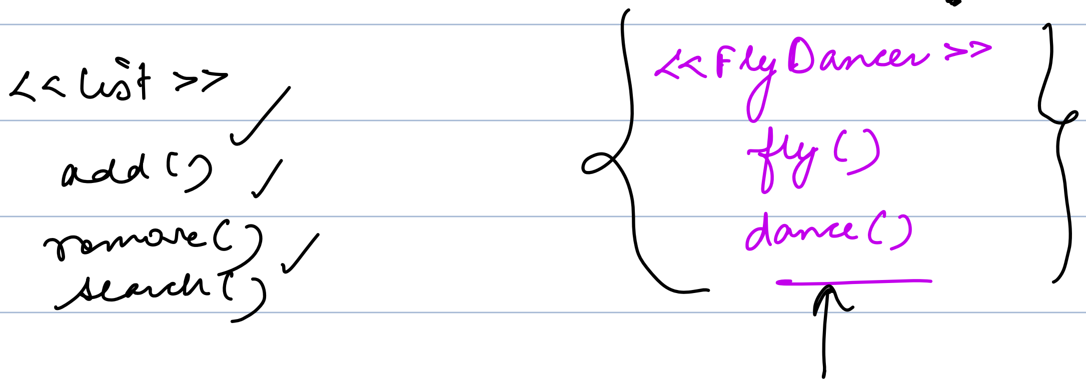


chosen UPI —— getcard details

```
CardUtility {
    doSomething( PaymentMethod ref ) {        card -
        CCPM ref2 = (CC) ref
        ref2.getCardDetails();
    }
}
```

Interface seggregation Principle

Some Birds can fly()
Some Birds can dance()
All the birds who can fly can also dance & vice-versa.

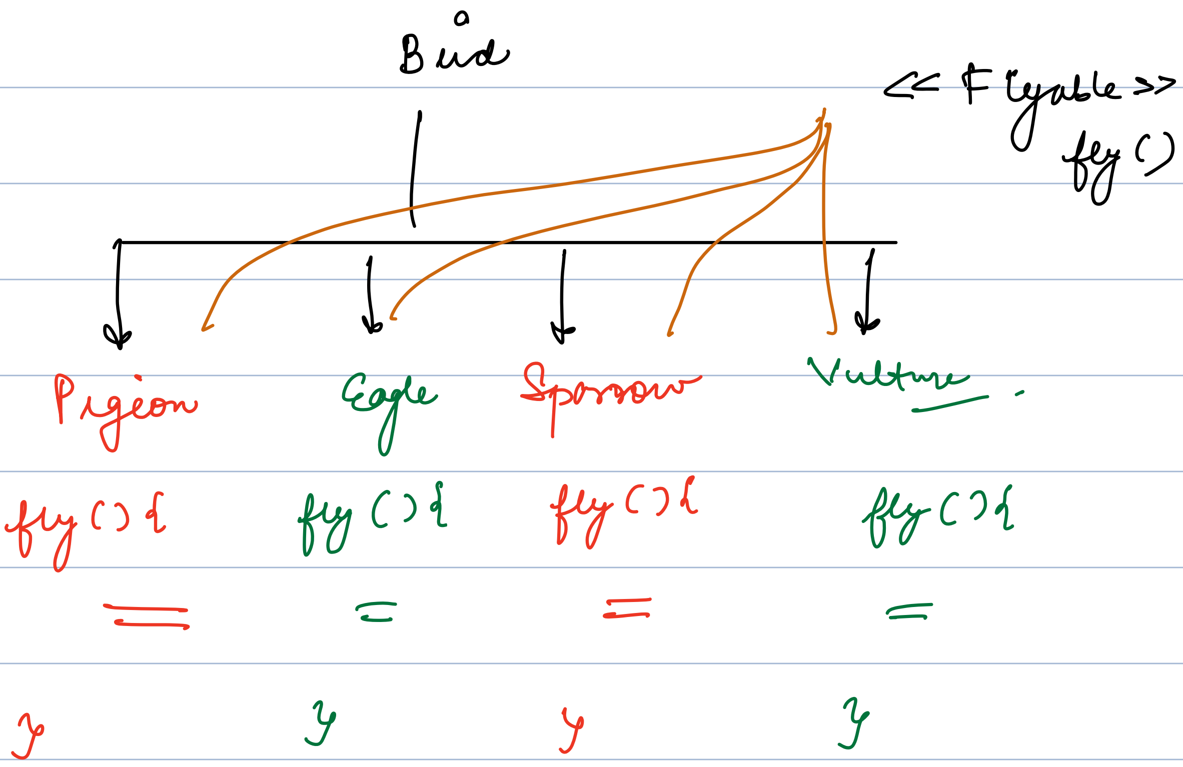


- Interfaces should be as light as possible

less methods

- Ideally interfaces should have one abstract method.

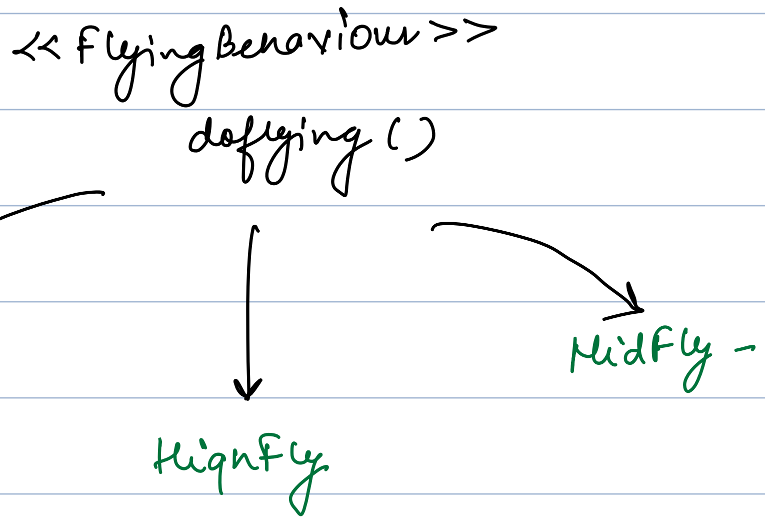
Dependency Inversion Principle



```
class  LowFlying {
    makefly () {
    }
}

class  HighFlying {
    dofly () {
    }
}
```

```
Pigeon {
    ~~LowFlying~~ x = new ~~LowFly~~();
    fly() {
        x.~~makefly~~();
    }
}
```



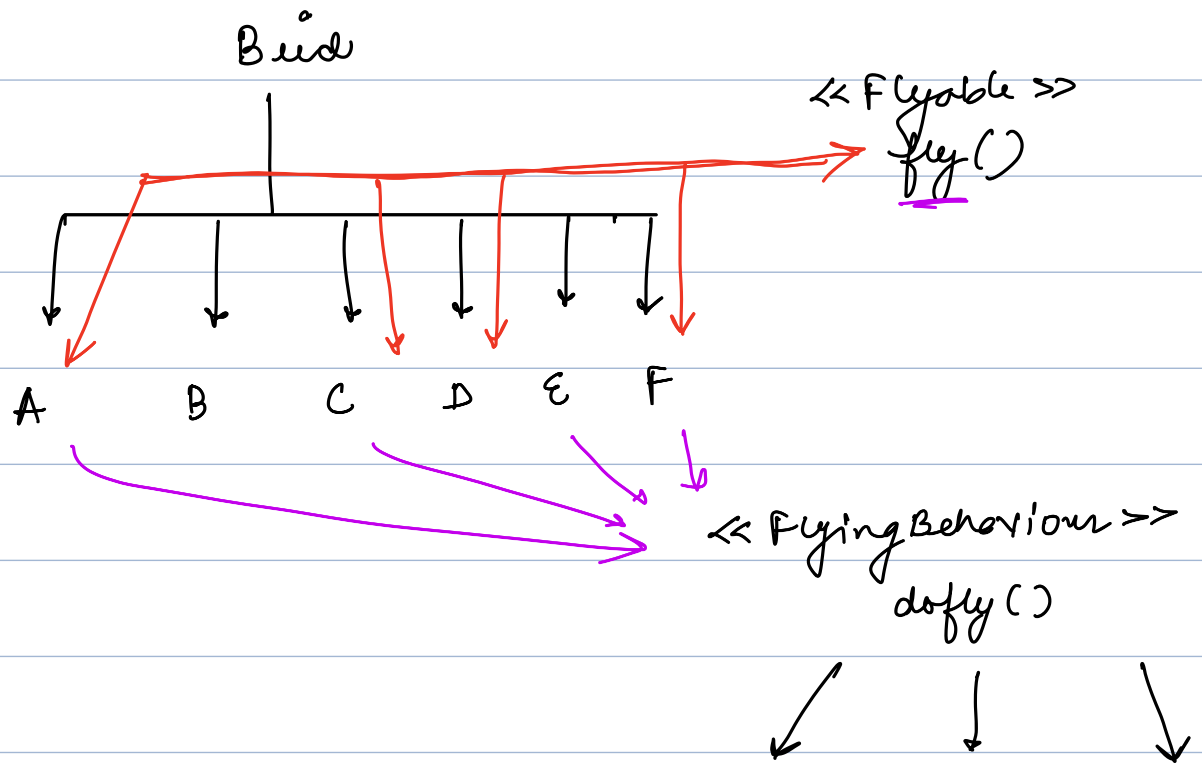
```
Pigeon {
    FlyingBehaviour fb = new LowFly();
    fly() {
        fb.doFlying();
    }
}
```

$$\begin{tikzcd}
\text{Pigeon} \arrow[r] & \underline{\text{LowFlying}}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{Pigeon} \arrow[rr] & & \text{Interface} & \\
& \text{LowFly} \arrow[ur] & & \text{HighFly} \arrow[ul]
\end{tikzcd}$$

→ No 2 concrete classes should be directly dependent on each other, instead should be dependent via interfaces.

" code to interfaces not to implementation".


Bird
<< Flyable >>
fly()
A
B
C
D
E
F
<< FlyingBehaviour >>
dofly()