SOLID Design Principles – 1

Agenda

1. SRP
2. OCP
3. Build foundation for Lekon's

SOLID Design Principles

Software Design

foundation / value / guidelines / basics

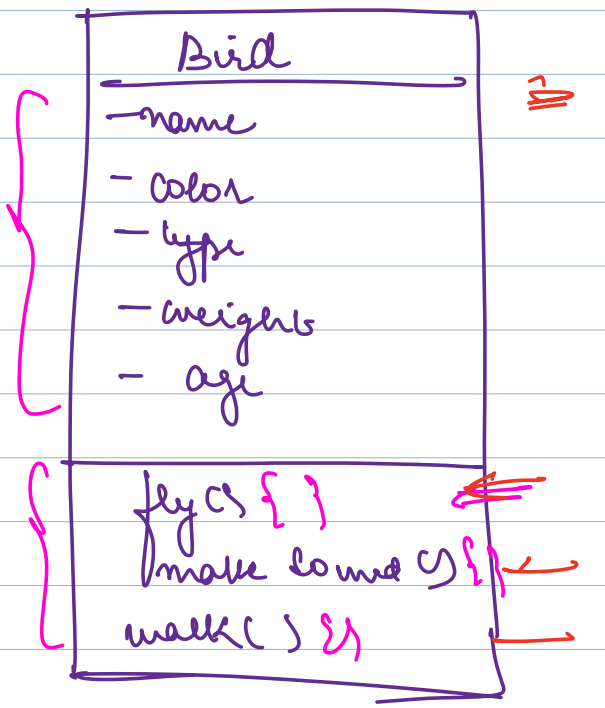
→ SOLID ⇒ Acronym for 5 principles that if followed shall lead to a great software design.

- → Extensible
- → Maintainable
- → Understandable.
- → Modular
- → Reusable

Design a Bird ⇒ Amazon Interview Q^n / VMWare

Req ⇒ { Assume you are building a S/W in which you have to store info about birds. Do the class diagram for such S/W. }

VO



⇒ VO

Client {

psvm() {

Bird sparrow = new Bird()

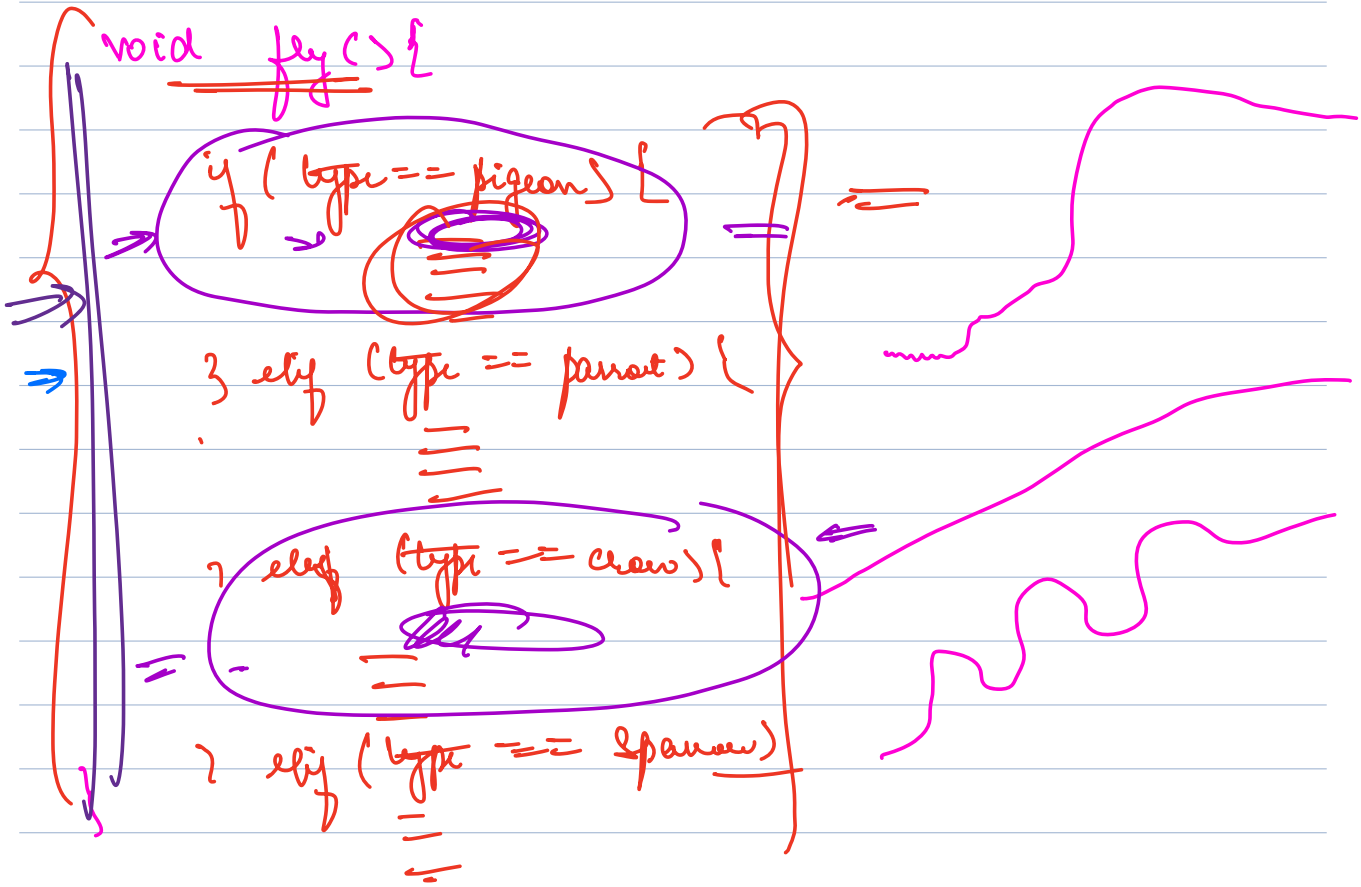
sparrow.type = ____

.weight = ____

sparrow.fly()

}

}



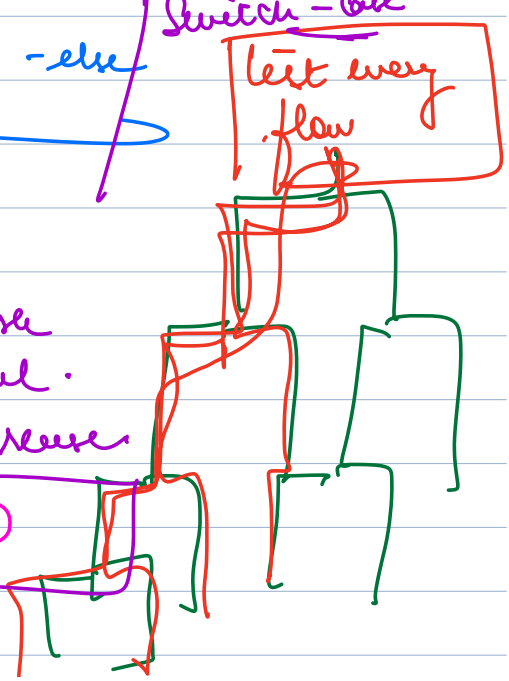
⇒ Difficult to maintain

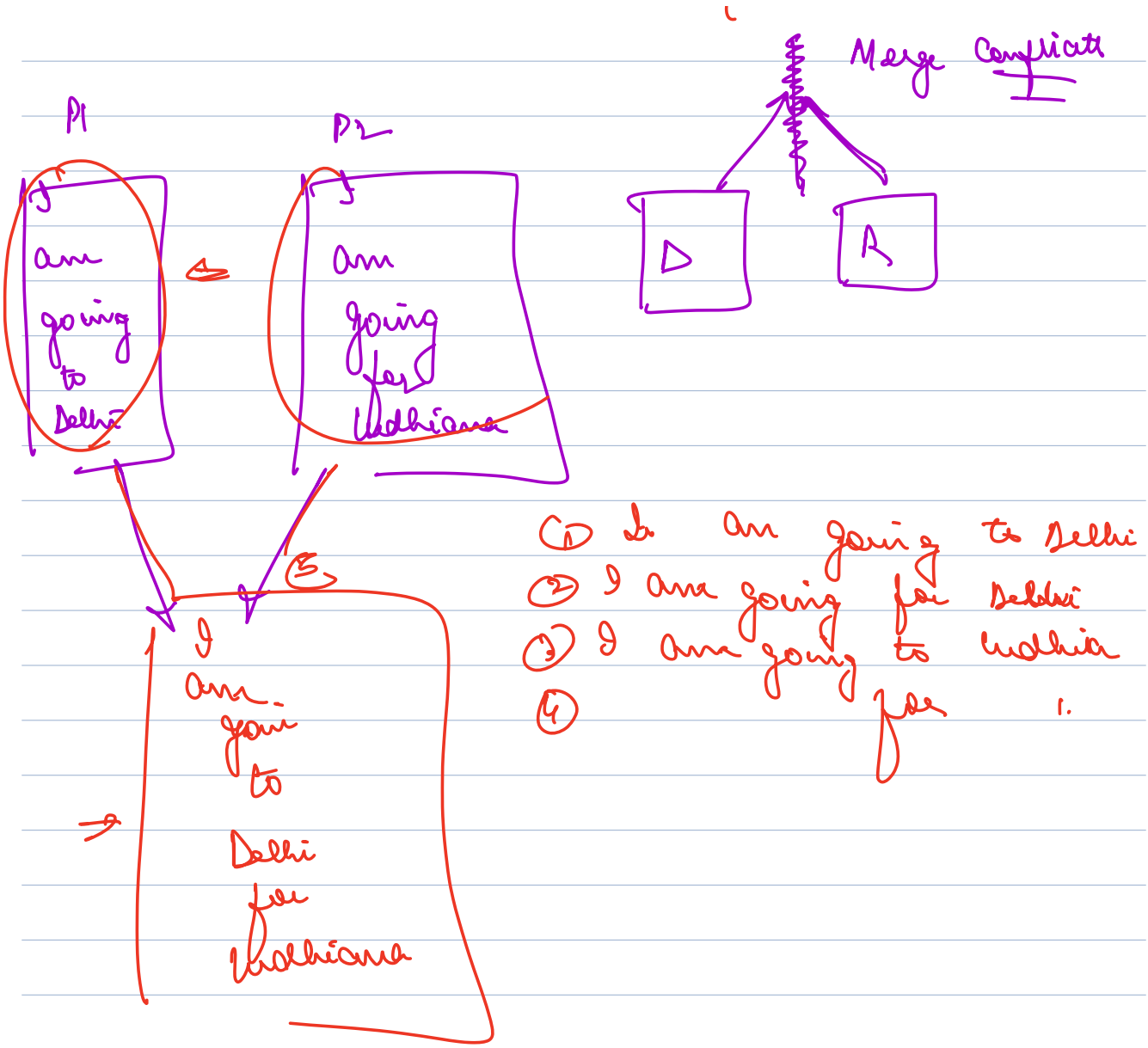
Problems due to too many if-else

Switch-case

Test every flow

1. Understandability
2. Testing
3. Diff for multiple people to work in parallel.
4. Code dup / less code reuse
5. Violates (S) of SOLID




P1
P2
I am going to Delhi
I am going for Ludhiana
I am going to Delhi for Ludhiana
Merge Conflict
D
A
① I am going to Delhi
② I am going for Delhi
③ I am going to Ludhiana
④ I am going for Ludhiana.

⇒ SINGLE RESPONSIBILITY PRINCIPLE

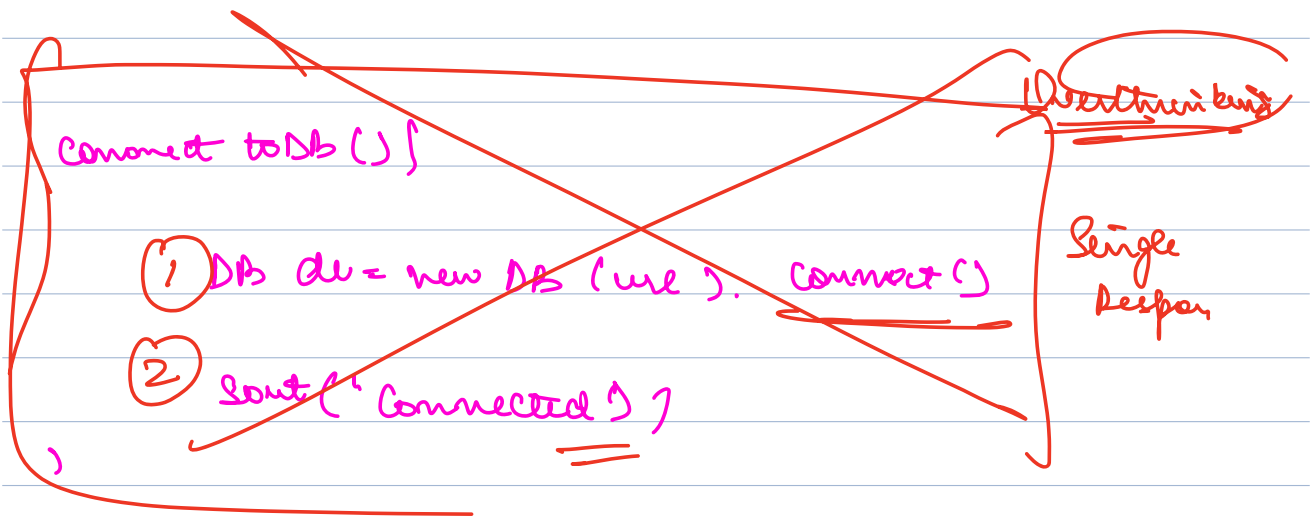
→ Every code unit (class / method / package) in codebase should have exactly 1 responsibility

⇒ it should be having code to do only 1 thing

⇒ exactly one reason why you might want to change code in that unit

fly() ⇒ how every bird flies

→ sparrow flies
→ crow flies
→ pigeon flies
→ penguin flies.



→ LLD is very subjective
→ If you are able to understand tradeoff of your design, you're good.

How to identify SRP violations

(a) Method with multiple if-else

→ Not always bad
→ If the if-else are a part of business logic that's okay.

(b) MONSTER METHODS ⇒ Affects reusability

→ Method which is huge
→ if it is doing too many things
→ that has code which does a lot more than what its name expects it to do.

Save To Database (User , Database obj) {

Generating a query

String q = " ____ "
q += " ____ "
q += user.name.

q += user.email

creating a conn

db = new DB().get()
db.set ConnUrl()

saving

db.save(query);

}

XYZ {

generateQuery(______) { }

Create DB Conn(______) { }

saveToDatabase(______) {

generateQuery()

Create DB Conn()

db.save()

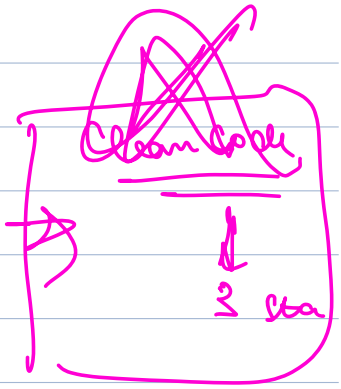
}

}

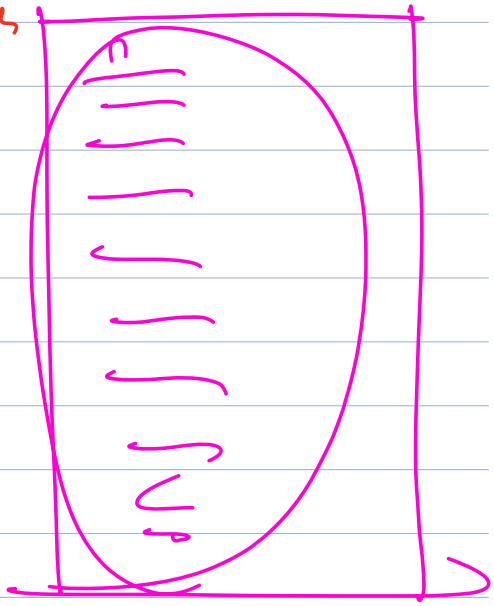
responsibility ⇒ how to do something

(C) Commons / Utils packages

Utils → discouraged
→ end up becoming a garbage store for every class you don't know where else to put



util . collection
util . date.util
util : Calendar
util . common
util . io



SRP → Every code unit → exactly 1 resp.

→ How to identify :
① y - else
② whenever
③ utils / common

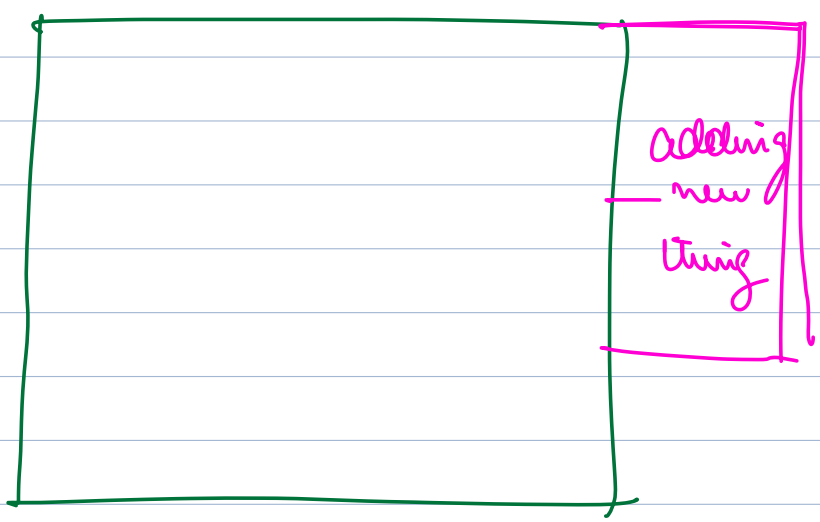
Open Close Principle

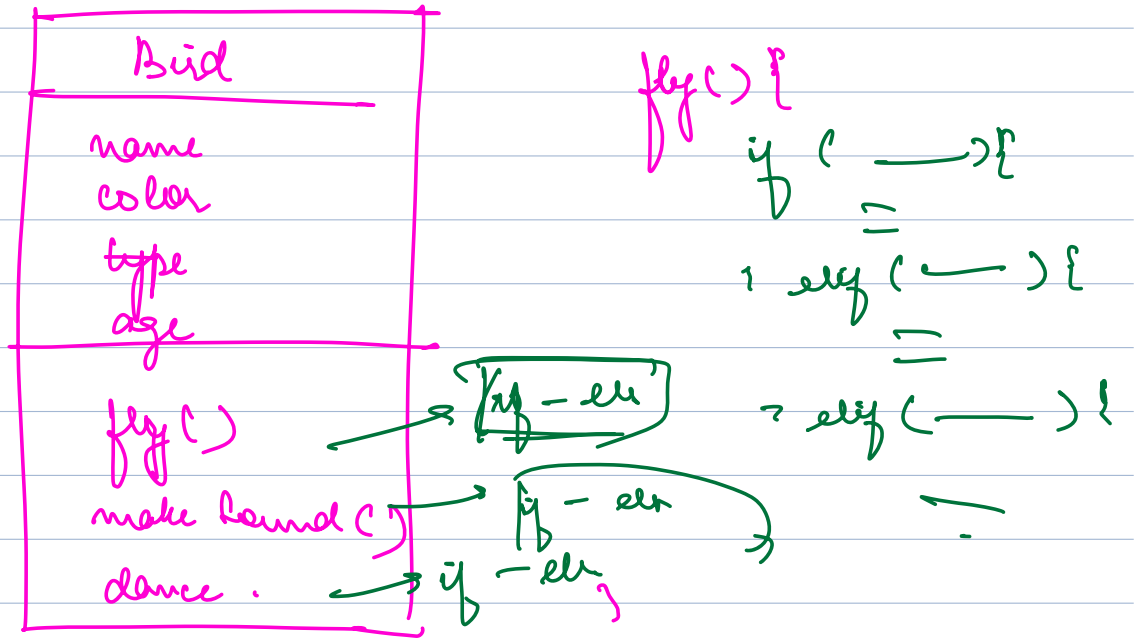
→ A codebase should be open for extension but closed for modifn

→ My codebase should make it easy to add new features.
But adding new features shouldn't require making much changes to already existing code base. Instead you should add new classes / functions etc

(1)

(2)





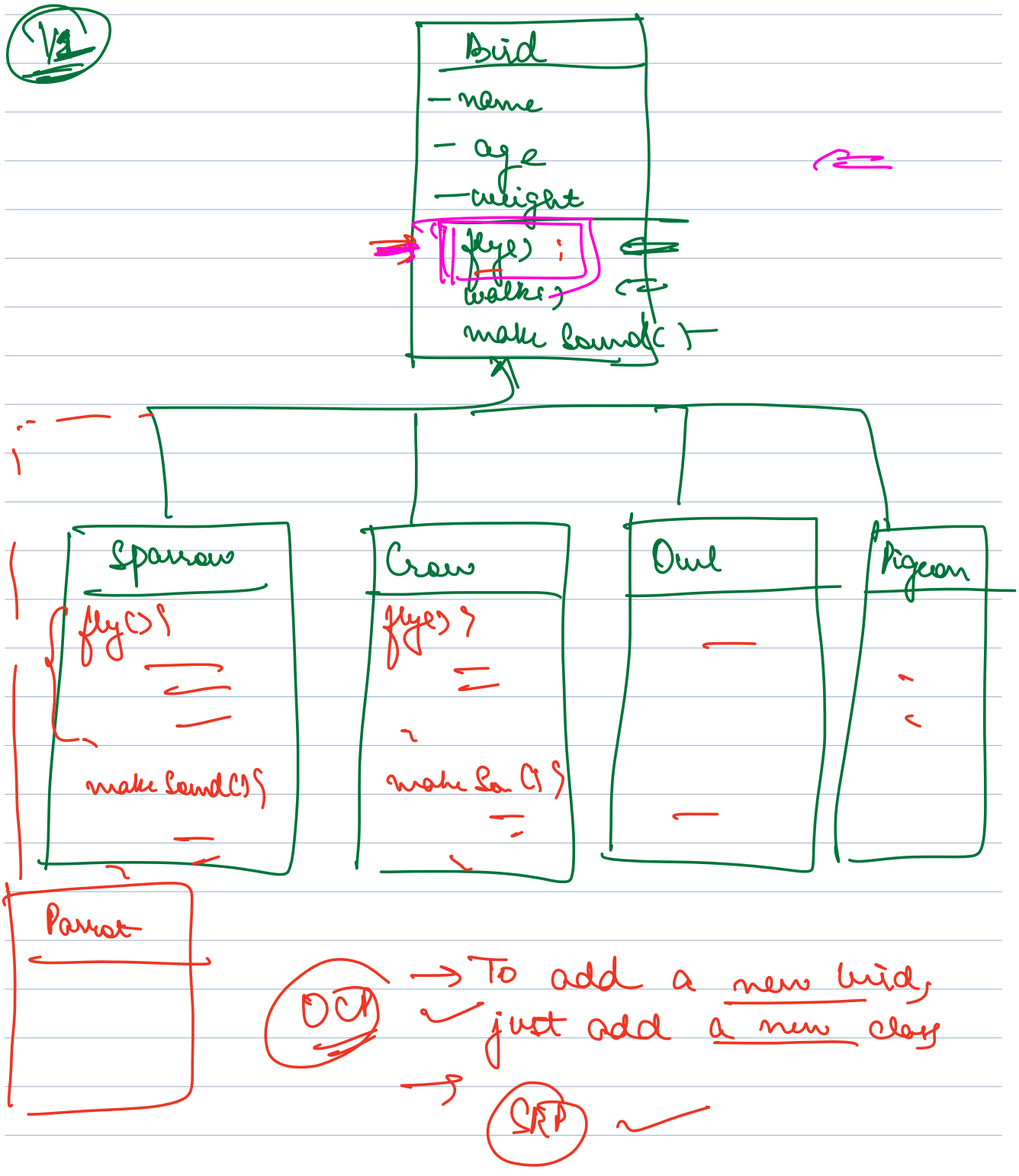
⇒ New feature req: Support a new Bird: Peacock

Why OCP

→ Testing

→ ~~Test~~ Regression

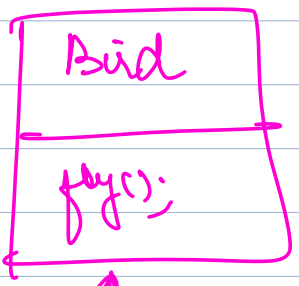
↳ when because of a new feature already existing feature breaks up.

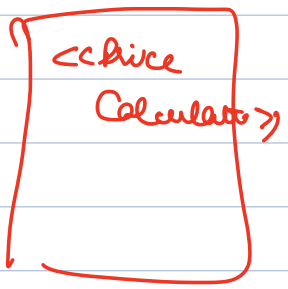


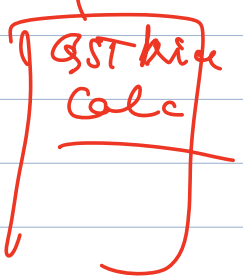
OCP → To add a new bird, just add a new class

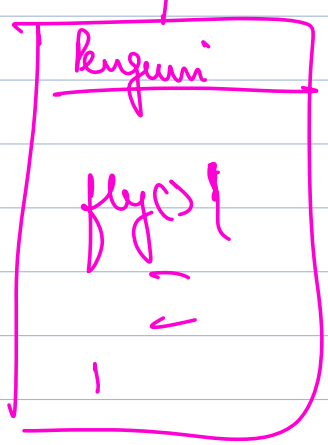
SRP

⇒ Req : Penguin can't fly / Ostrich / Kiwi





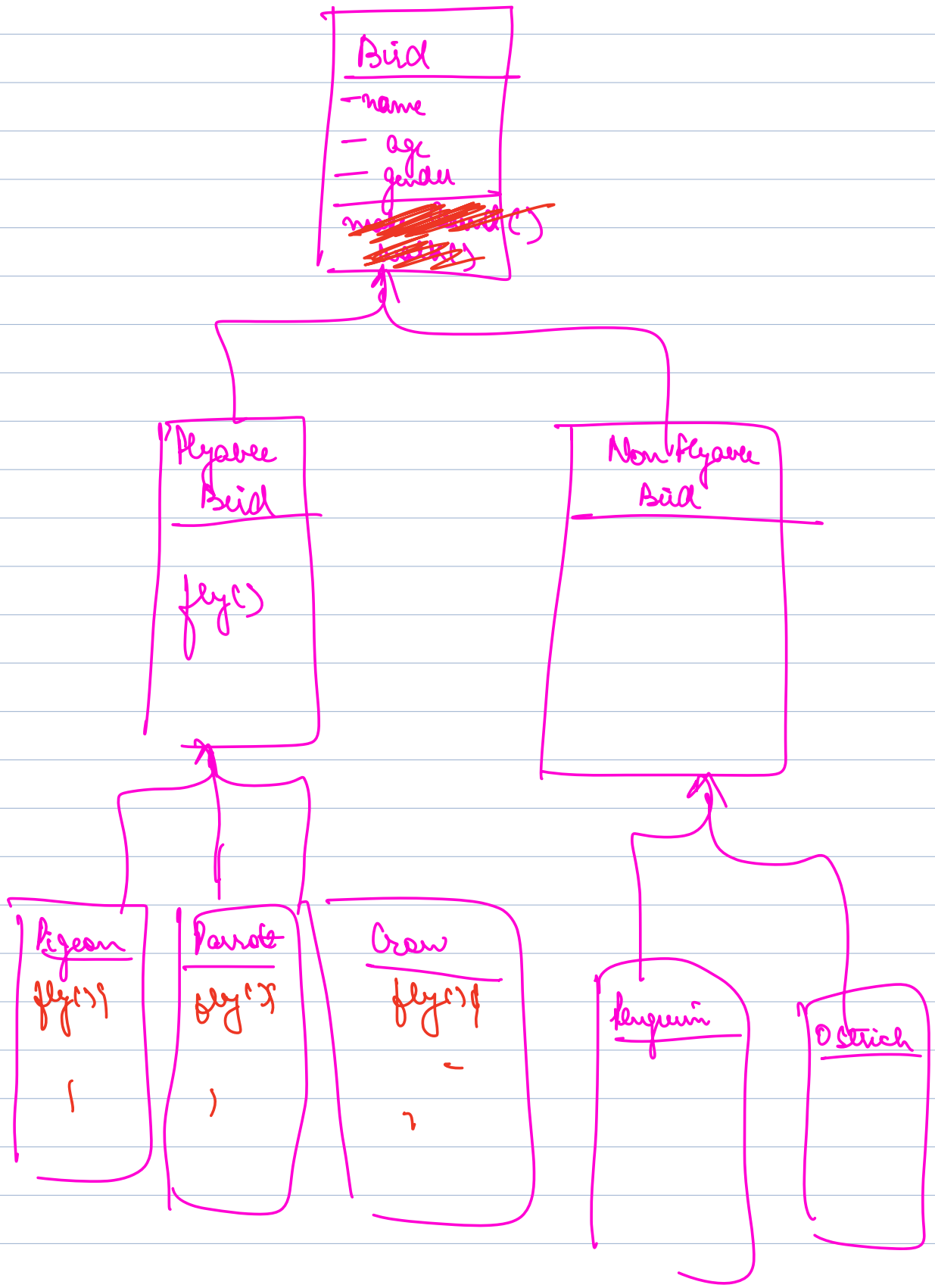


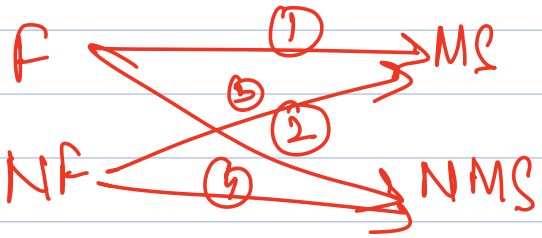


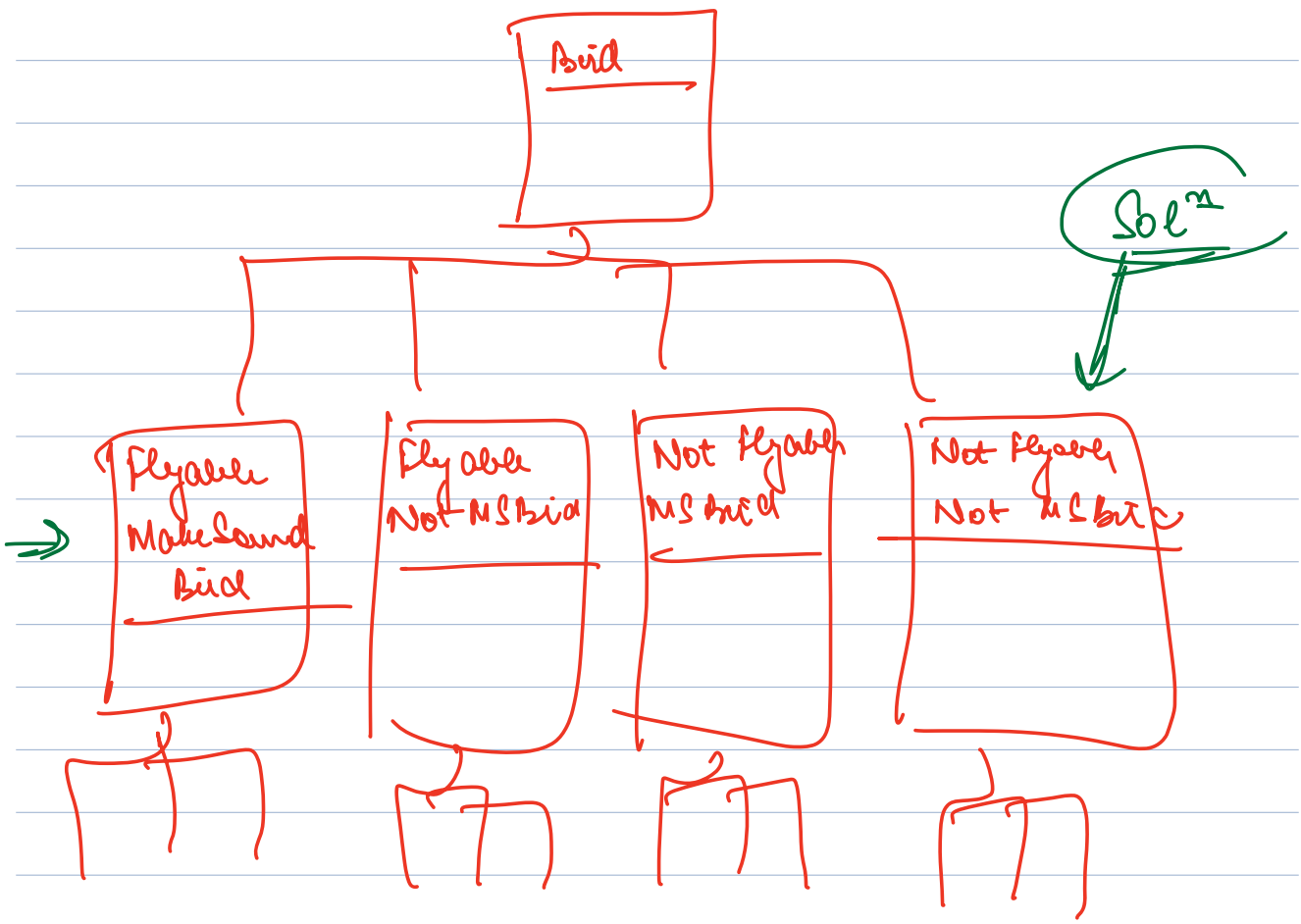
① Not implement fly in penguin

② Throw exception from fly in penguin().

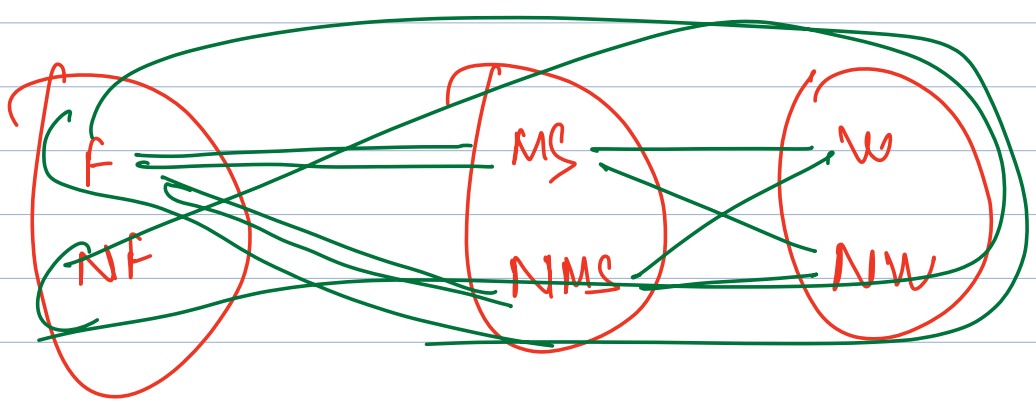
③ Create 2 child classes of bird: Flyable Bird, Non-Flyable Bird


Bird
-name
- age
- gender
FlyableBird
fly()
NonFlyableBird
Pigeon
fly()
Parrot
fly()
Crow
fly()
Penguin
Ostrich

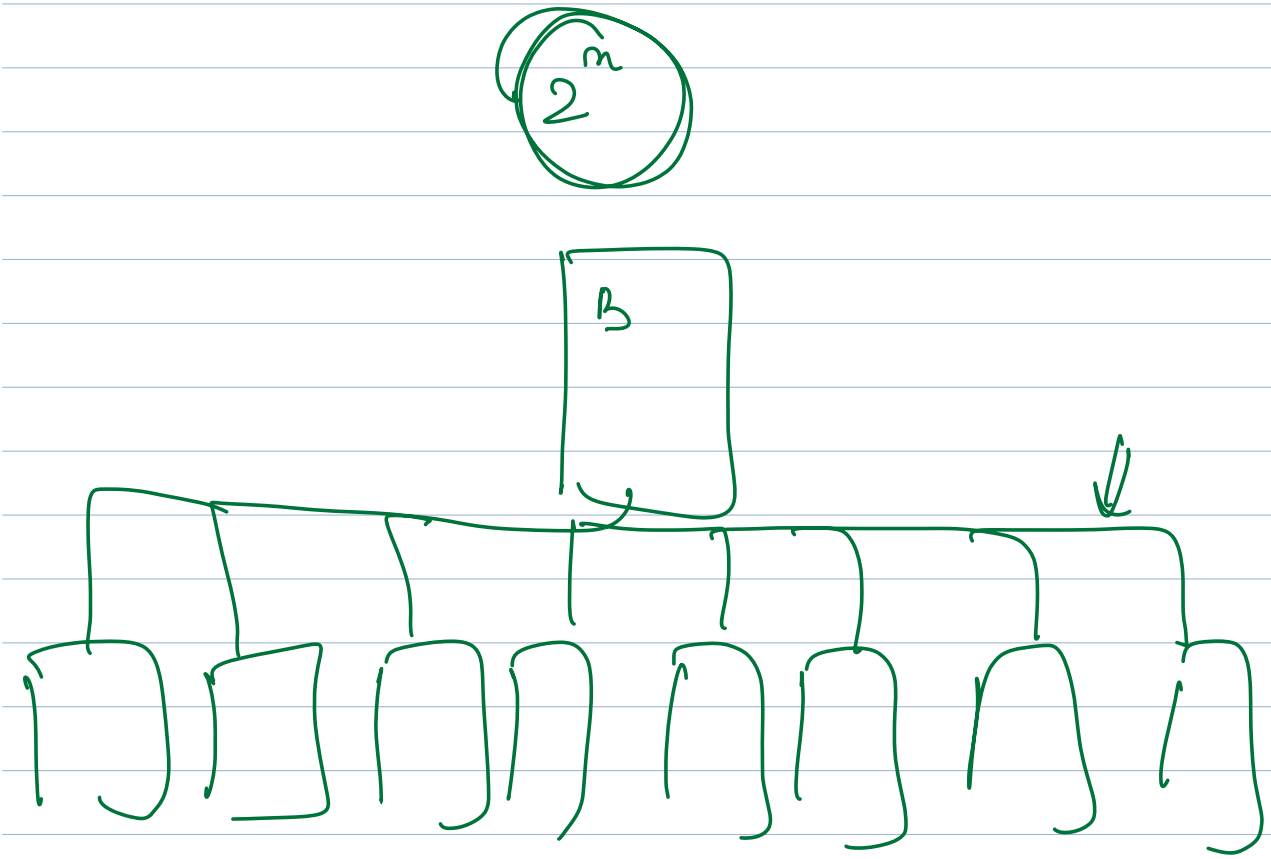

F
NF
MS
NMS
1
2
3
4


Bird
Soln
Flyable
MakeSound
Bird
Flyable
Not MS Bird
Not flyable
MS Bird
Not flyable
Not MS Bird

8


F
NF
MS
NMS
W
NW

6 9 8



$(2^{10}) \Rightarrow 1024$

Class Explosion

Wed ⇒ Remaining OOI

interfaces polymorphism abs class static

40, 41 ——— 45

Strong Hex