SOLID - II

SRP

OCP

Design Patterns

→ Creational

Case Study

→ Bind

V0

fly() {

v1


```
if (type ==)
else if( parrot
...
}
```

O/C
SRP

Solutions

* dummy

* return null

* throw an exception

List<Bird> = List.of()

Parrot()

Penguin()

for each bird:

bird.fly()

if bird is flying:

doThis()

if bird is Penguin()

doThis

Liskov substitution principle

→ in order to handle any sub classes, I should not have to do any special handling

Formal

↳ Penguin p = new Penguin()

Bird (pT) = (Bird) p;

plt.plot() vs sns.plot()

Database

- query

MySql
query

Mongo
query

CIA
- query

→ try → throwing an exception

Real life

Diesel Car

Start engine

Start
engine

Substitution

→ Subtyping

Electric Car → Car

Car.startEngine()

if I substitute Electric → Car
the it should behave normally.
special handling

Fly ?

.fly()

→ .sitOnBench()

9

Implement all methods

Else end

Choose another approach

Fix LSP

Bird
Sparrow
fly()
Parrot
fly()
Penguin

Plate of Bird[]:

List<Bird> = [...]

bird.fly()

---

(2)

multi-level

→ behaviour to classhierarchy

→ Swimming

	S	R
P	x	✓
P	✓	x
Due	✓	✓

→ Swimmable Flyable Bird

→ Non-swimmable Flybird

→ Class explosion

Typing behaviour to class structure

interface → State X

Java interface

→ blueprint for behavior

→ static

→ definitions of your methods

interface

→ Use to describe structure

→ multiple classes

Interfaces

→ Flyable

→ Swimmable

multiple inheritance

✓ Create a list of all birds

List<Bird> birds = {new Dove(),
new Pigeon(),
Penguin)

penguin.fly()

Create a list of all birds that can fly

List<Flyable>

<Swimmable>

Set all birds to fly

LSP

List<Bird>

instance of

Interface Segregation

No fat interfaces

Bird + Flyable ?

Skip

not on null

thick

Interface segregation

Interface segregation

→ SRP compliant

→ Extensibility

Functional interface

Single abstract method
(SAM)

Keep relevant methods

Iterator {

next()

→ high

cohesion

hasNext()

SOLID

fly()

eat()

① High cohesion

② Low coupling

6:01 – 6:05

10: 35

(2) = 10'

add(int a)

add(a)

autocasting

autoboxing

Parent Child

box derived

subclass

super

LSP — interfaces

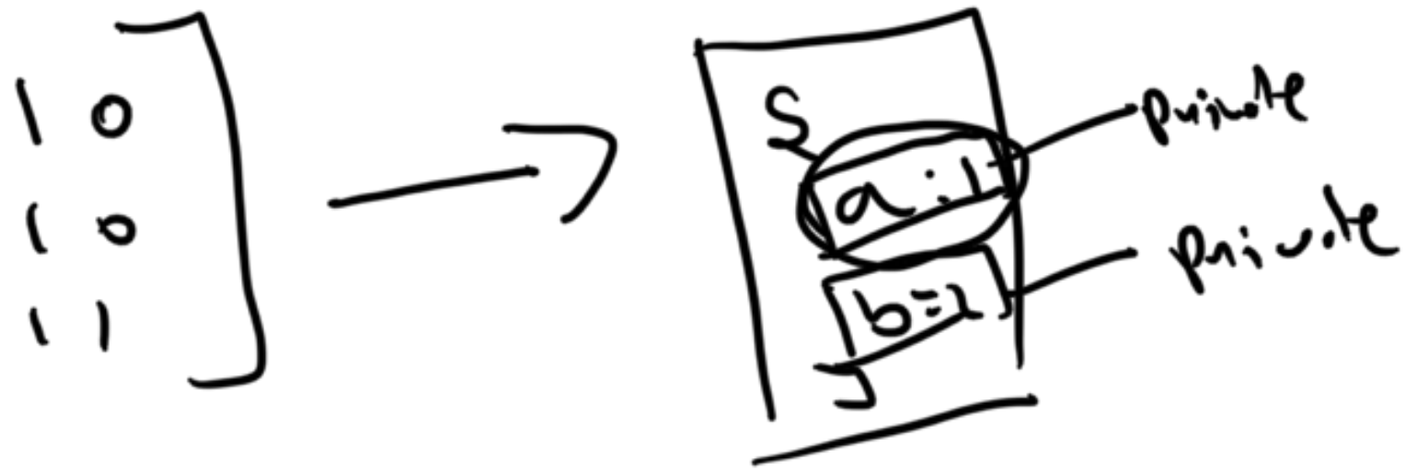
ISP

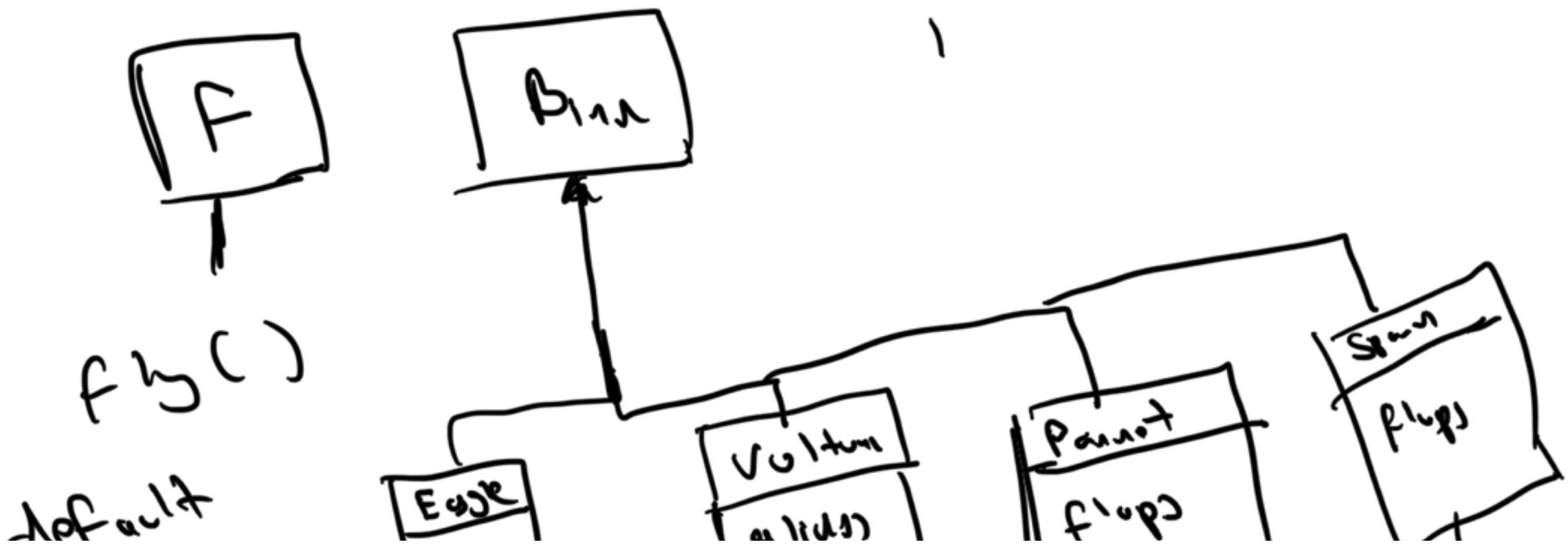
---

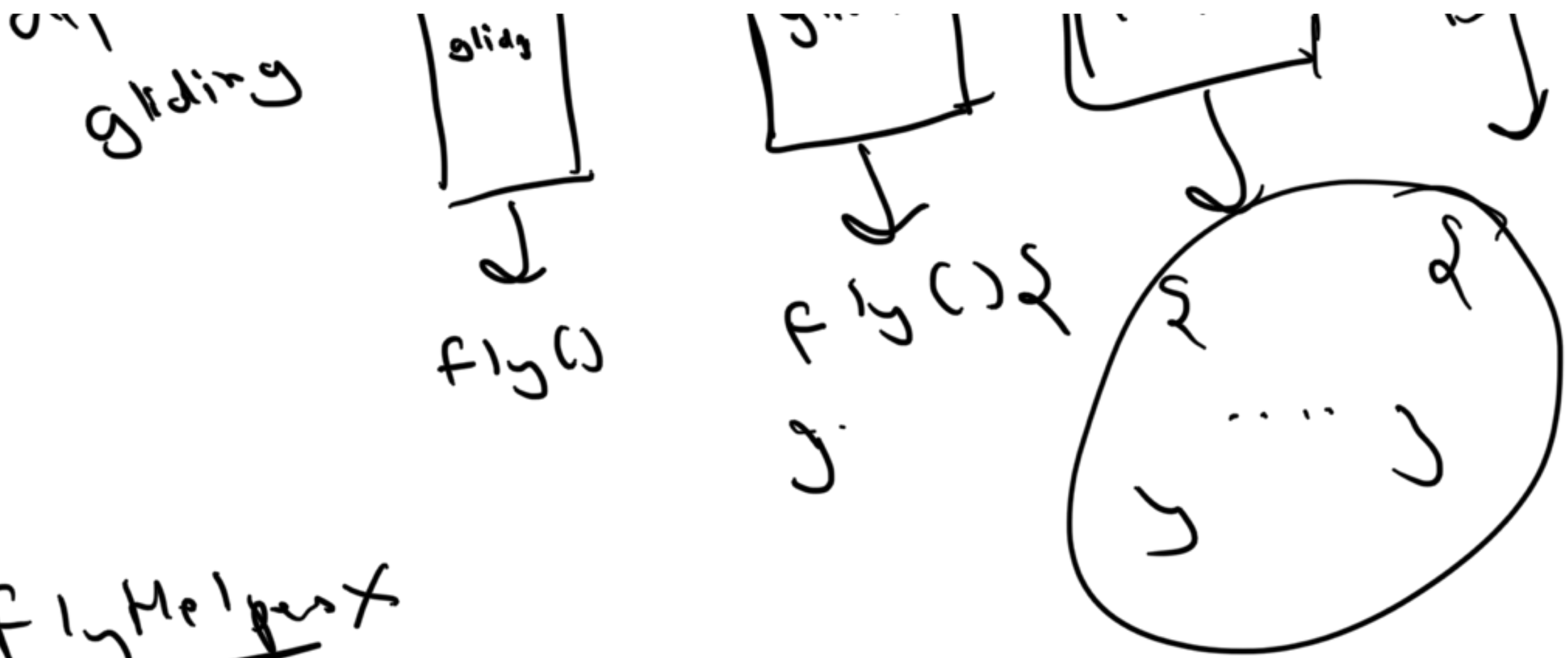
interface

class

Memory allocation






Gliding
fly()
fly()
SRP
Glide
Flap
code
duplication
Better solution

→ Gliding Behaviour

→ Flapping Behaviour

Eagle extends Bird imp. Flyable {

GlidingBehaviour
ctn C GlidingB)

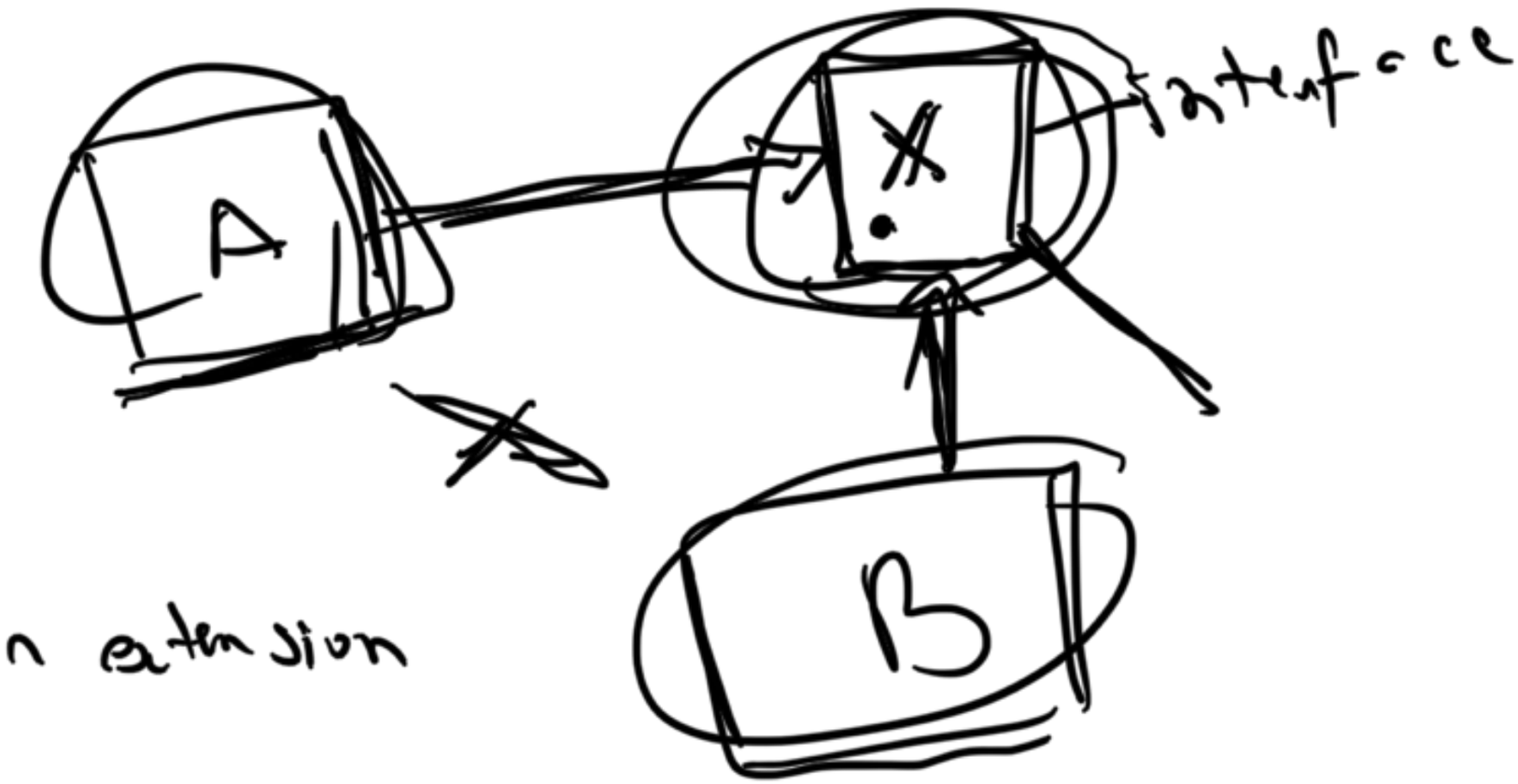
fly() {

GlidingBehaviour.makefly()

Dependency inversion

① High level modules (Parent) should not depend on other concrete classes.
Both of them should use abstraction.

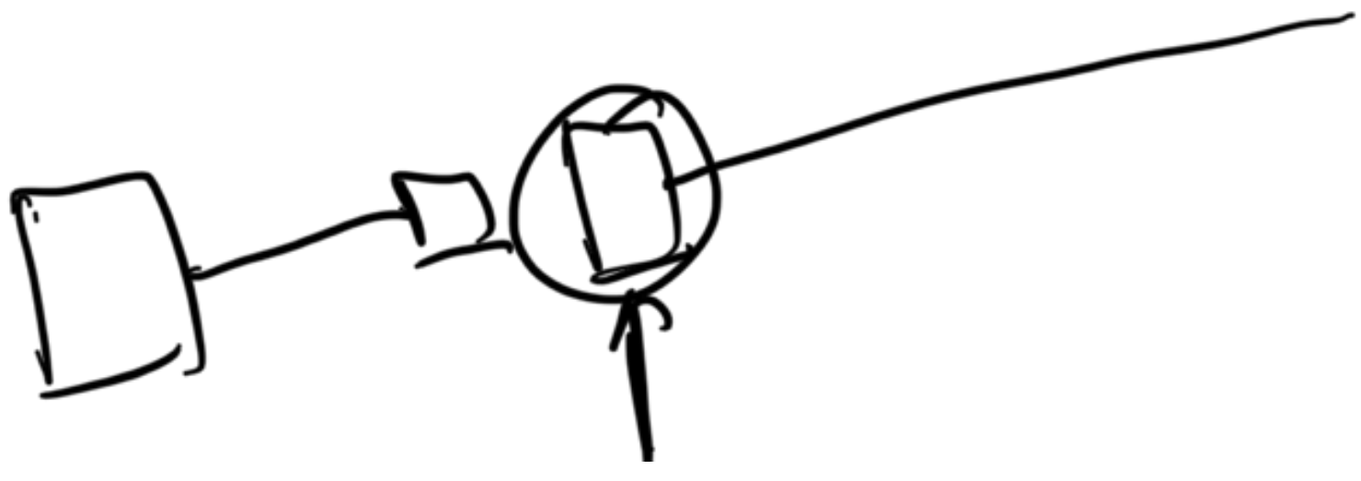
tightly coupled



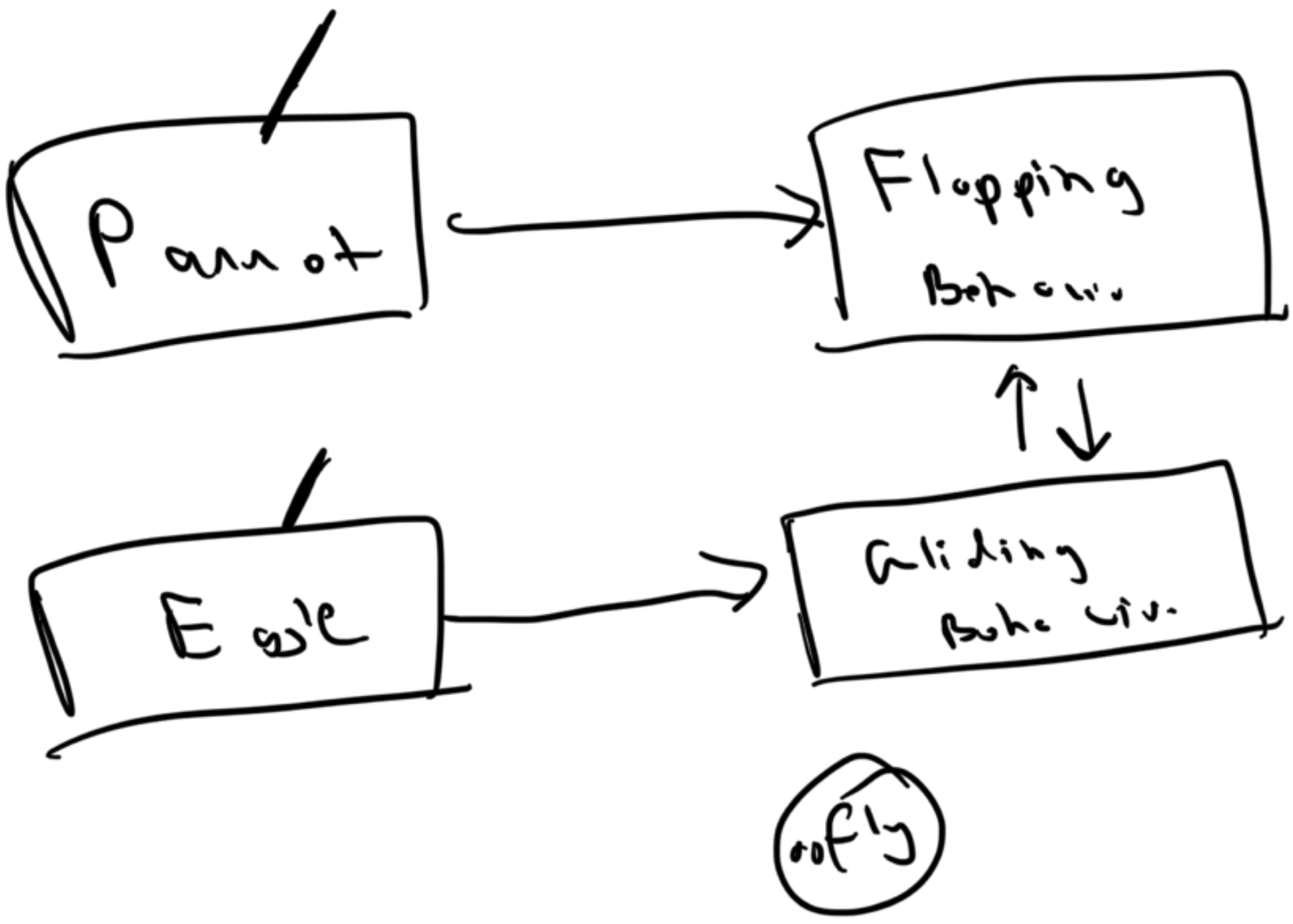
Open for extension

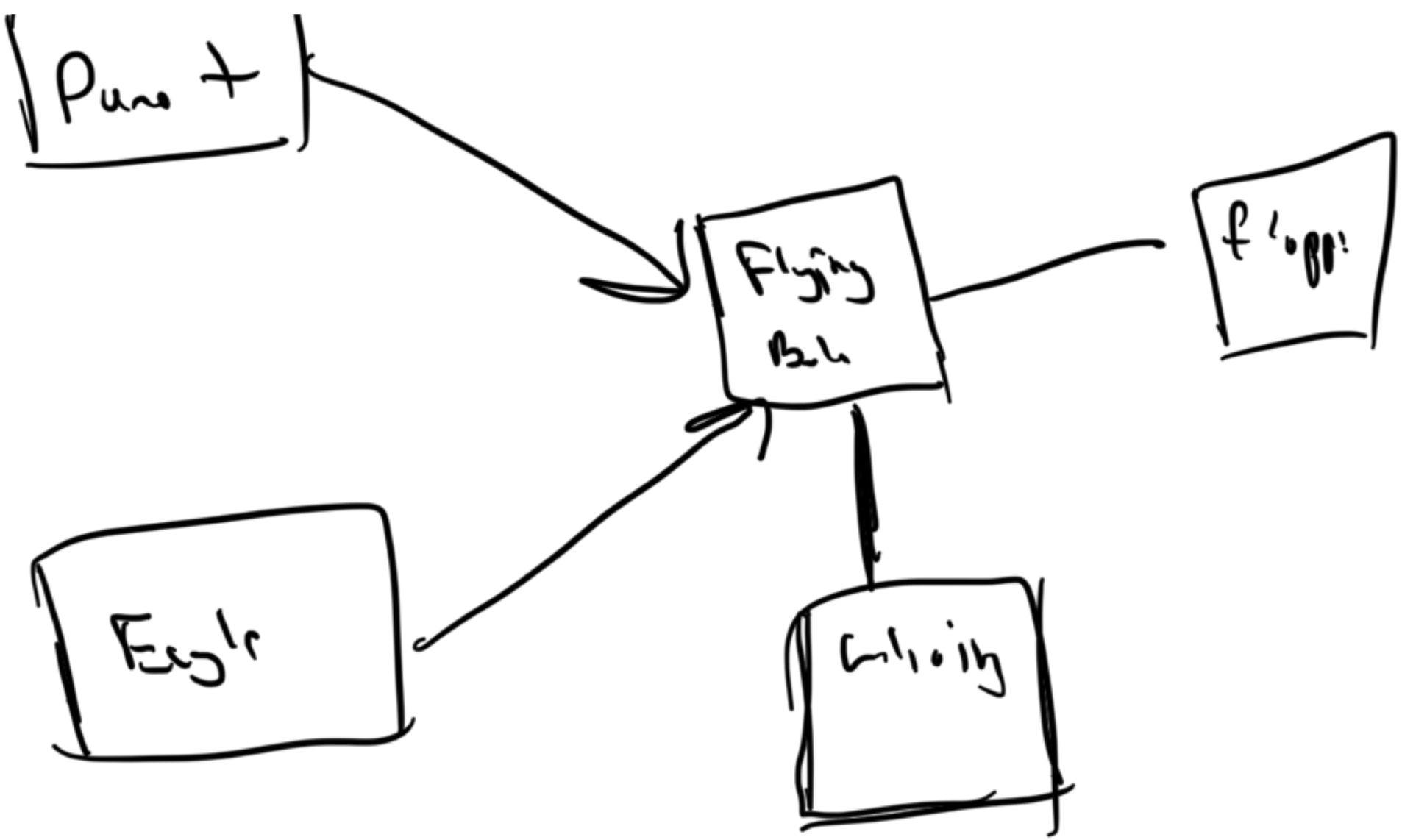
Close for modification

Solder to



middleman





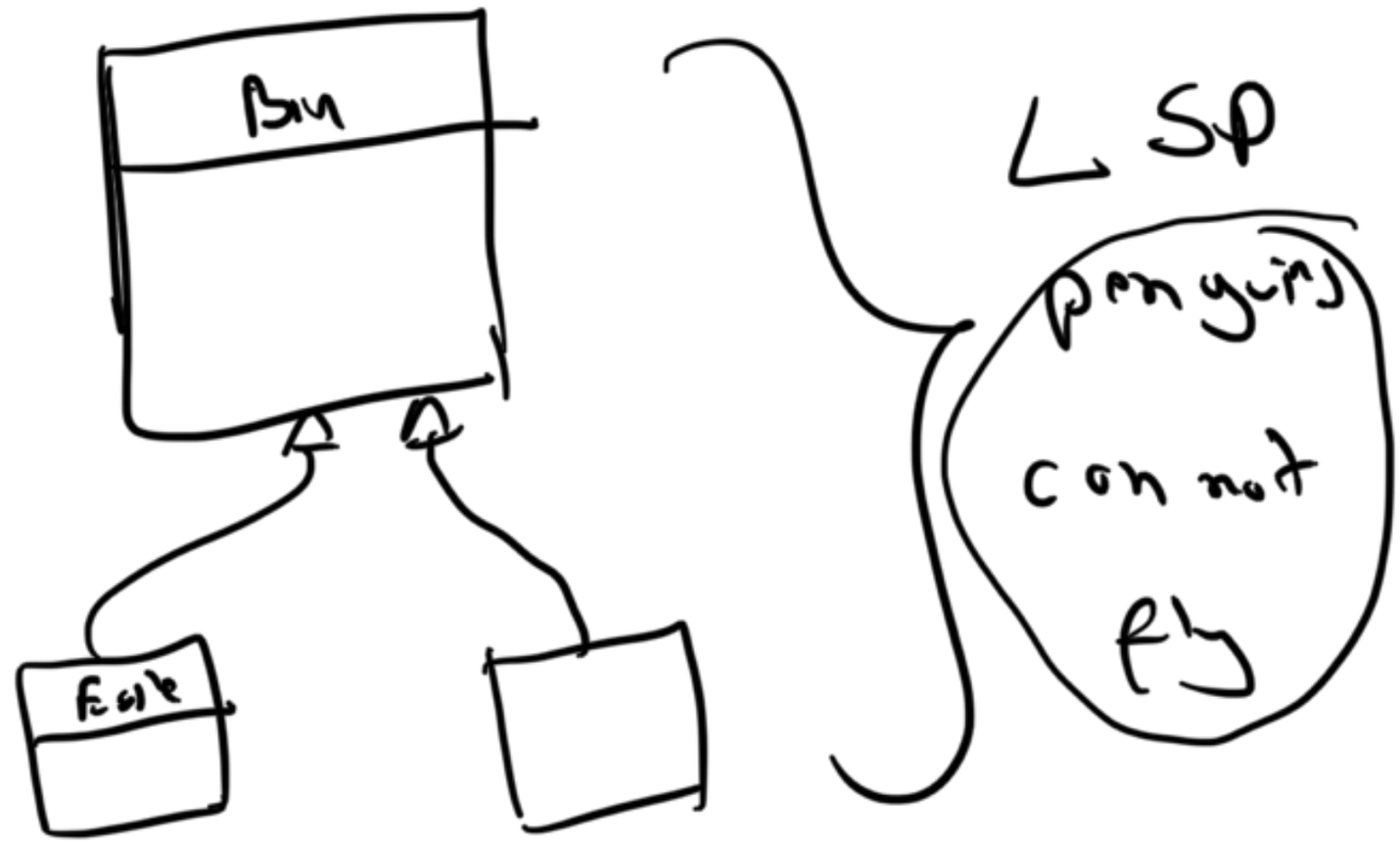
Loose coupling

VO

SRP

OCP

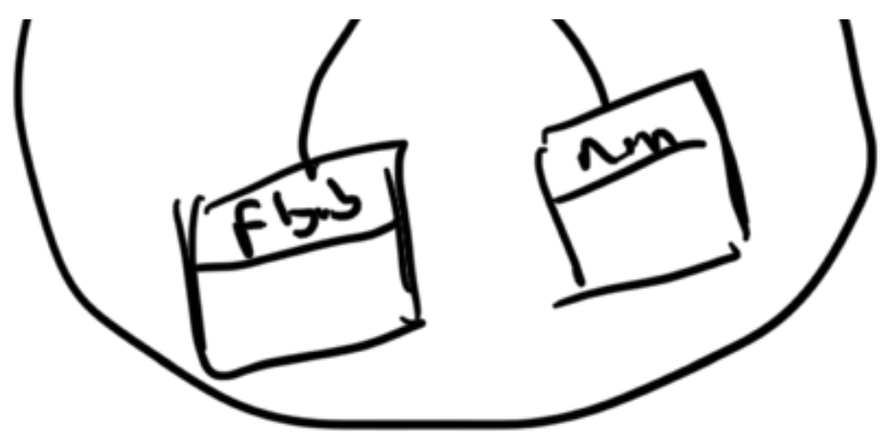
v1



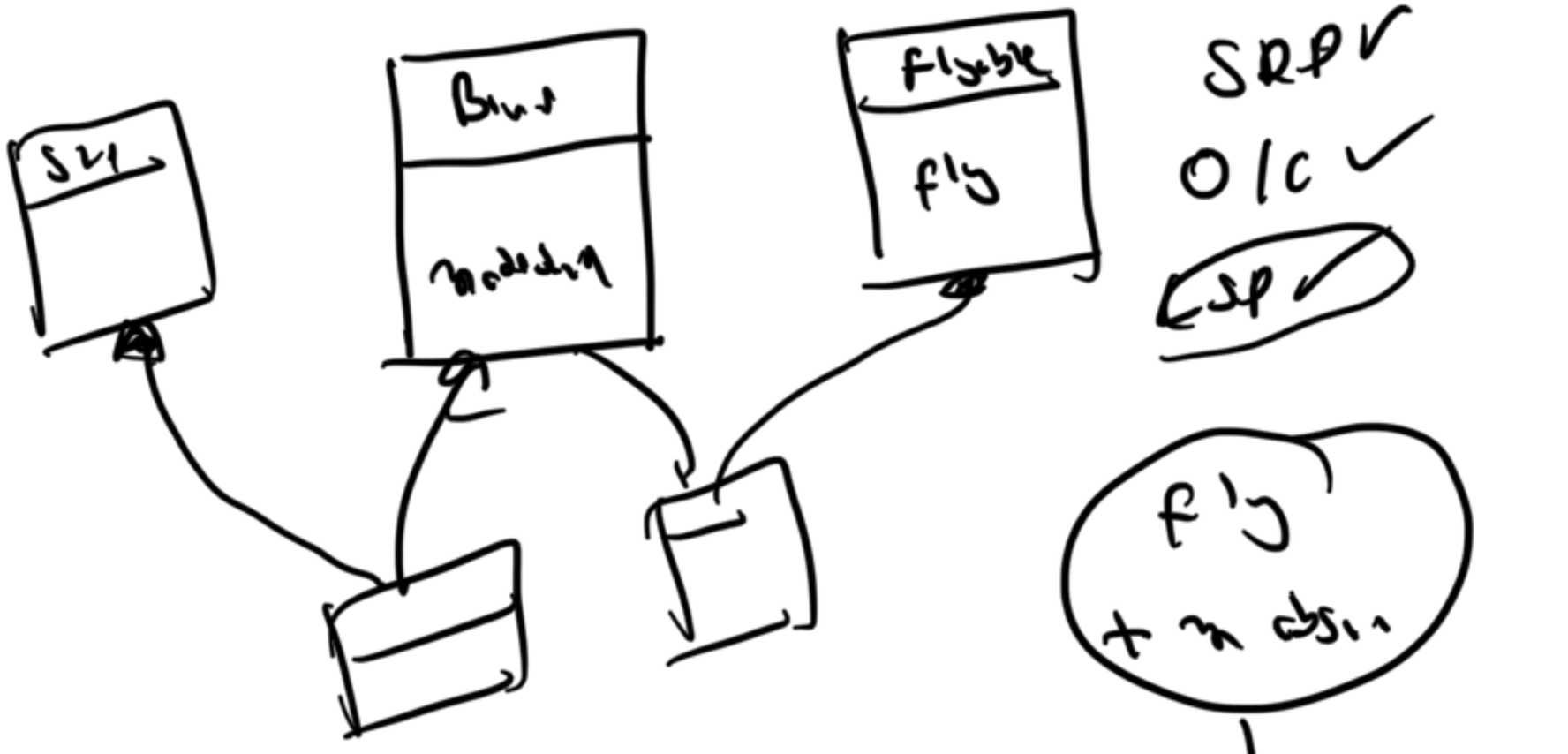
v2

→ Class explosion

→ Swimmable flyable



v3



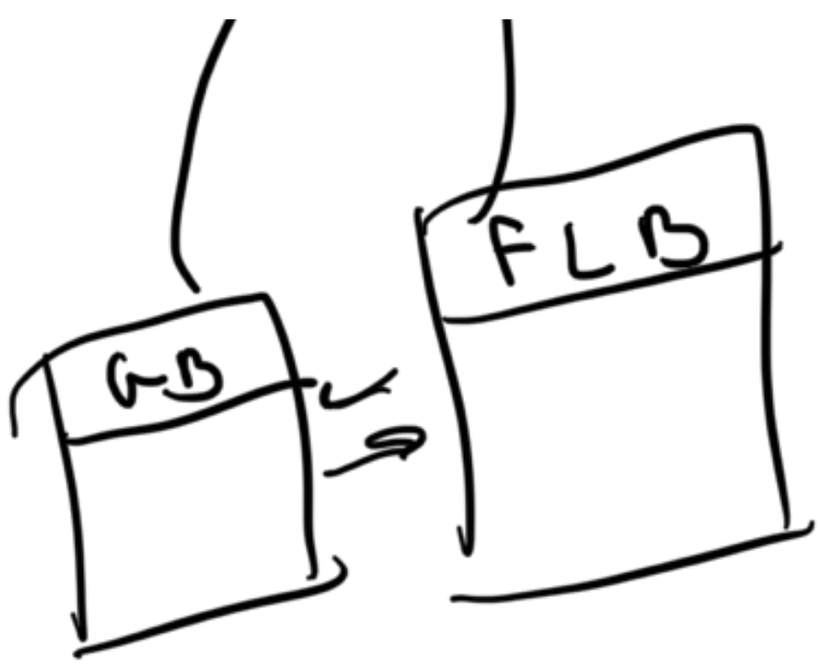
v4

SRP

Ocp

Con crete ⟷ Con crete

V6



SOLID → loose coupling

- SRP
- OCP
- LSP

→ DI

high cohesion

→ ISP

SOLID

SRP — only one reason to change

OCP — open for extension

— closed for modification.

↳ LSP — no special handling for any subclass

→ ISP — lean interfaces

→ DI — concrete classes should not depend on each other

SOLID