Design Principles

Stay focused on the big picture!

S SINGLE RESPONSIBILITY
(Keep it Simple Silly)

O OPEN CLOSED

L LISCOV'S SUBSTITUTION

I INTERFACE SEGREGATION

D DEPENDENCY INVERSION

D

Depend on the generalization (high level) & not the low level concreate classes.

COMPOSING OBJECTS PRINCIPLE

Composition over inheritance to achieve loose coupling & code reuse

I

Do not let your classes have dummy methods!

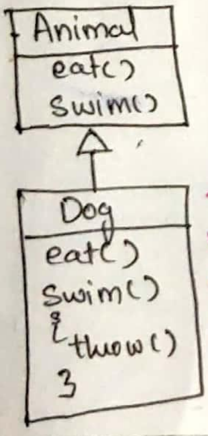
* A class should not be forced to depend on methods it does not use

No timepass! Do only what you have to do!

Others that are not solid

1. Composing objects principle
2. Dont repeat Yourself (DRY)
3. Encapsulate changes
4. Composition over inheritance
5. Program to interface not implementation
6. Delegation Principle

If a class S is a subclass of class B then S can be used to replace all instances of B without changing behaviors of the program.



- inheritance check
- rules to keep inheritance in check.

This is a violation as Dogs can't swim.

L

A class should ~~do one~~ have one task it is supposed to do.

By Having a class to do more than one type of activity we end up in complex (Cohesive) code.

S

You had one job!!!

The classes should be open for extension (inheritance or polymorphism) and closed to change.

O

- enables maintainability by providing loose coupling b/w stable & varying parts

Get ready to expand yourself.

Have a broad mindset.

{ DESIGN PRINCIPLES }

Liskov's Substitution Principle – inheritance checking principle

If a class 'S' is a subtype of a class 'B', then S can be used to replace all instances of B without changing the behaviors of the program.

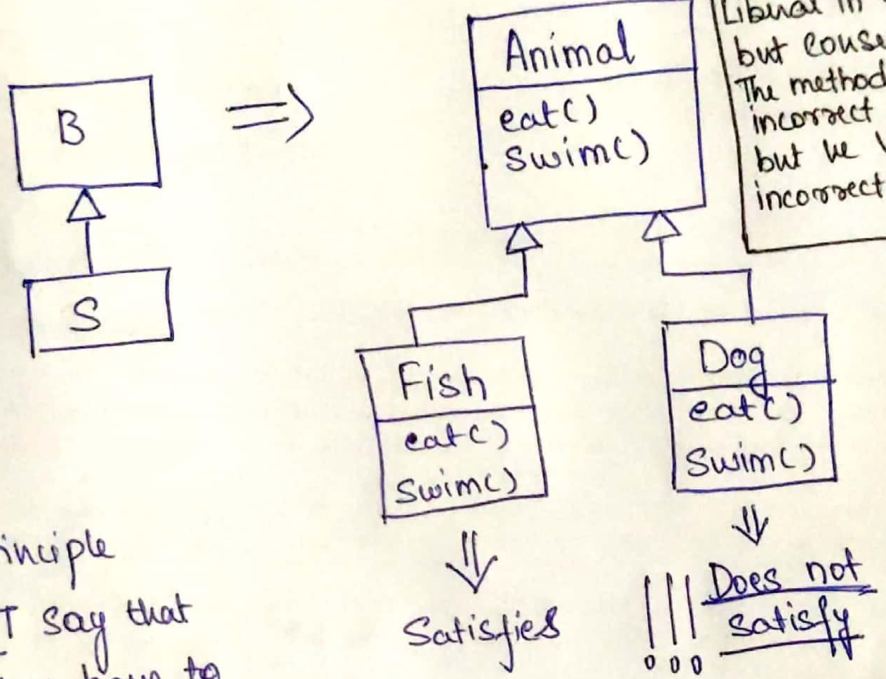
The principle helps to determine if inheritance has been used correctly.

If the expected behavior between the base class & sub class are different, then this principle has been violated.

The Robustness Principle

Liberal in the parameters that are accepted but conservative in what you send.
The method should be able to receive incorrect values (and act correctly), but be VERY careful in sending incorrect values itself.

Ex:



Behaviorally dogs cant swim! However, by adding swim method in the Animal class, you force the swim behavior to all the species of animals.

SOLUTION TO FIX

By using composition such behavioral fixes can be done. 'HAS A' relationship.

"The principle DOES NOT say that the subclass have to be replaced with its base class but, to check the correctness of inheritance you should be able to substitute the subclass with base classes. And if the behavior remains same then the inheritance check is passed!"

In the example above, ~~although both~~ the 'eat()' behavior ~~satisfies~~ passes the inheritance check as both fish and dog eat (although the details of the ~~behav~~ eating behavior, i.e, how they eat may differ).

Further: Pre Conditions, Post Conditions, Invariants

Precondition

What the method receives

Post condition

What a method returns.

To Satisfy the liscov's substitution principle........

The methods of the subclass should receive

① Same as what the method in the base class is expecting

(or)

② ~~And~~ it may also receive something more

SAME OR WEAKER

The methods of the subclass should return

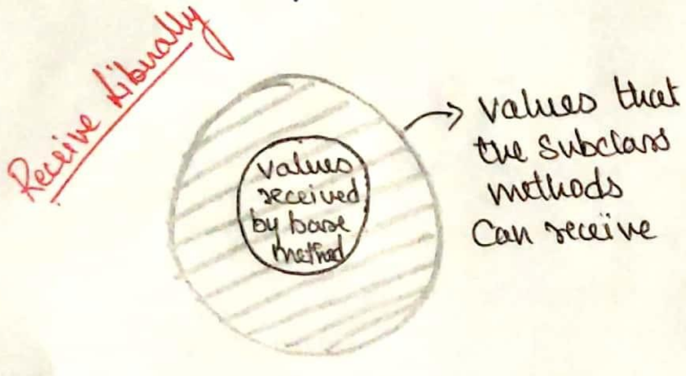
① values same as what the method in the base class returns

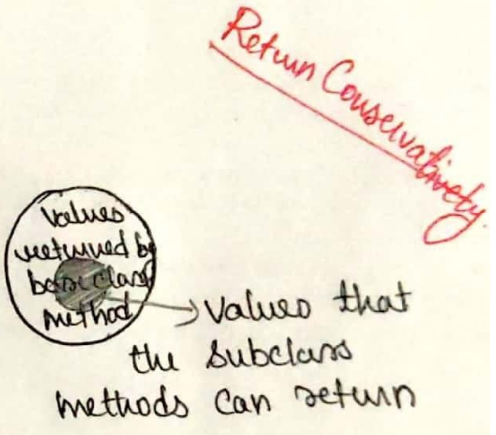
(or)

② subset of the values that the method ~~of~~ base class returns

SAME OR STRONGER

By following the above conditions your design would never deal with something unexpected.


Receive Liberally
values received by base method
Values that the subclass methods can receive


Return Conservatively
Values returned by base class method
Values that the subclass methods can return

The Robustness Principle

Be liberal in the parameters that are accepted, but be conservative in what you return.

(*) Invariants (Something that does not vary)

The subclass invariants has to be same as that of base class invariants.

Eg of invariant

Loop invariant is a condition that is true at start & end of every loop execution.

The same rule of invariants applies to the methods as well.

[My Thoughts: Maybe the invariants here refer to the ~~method~~ conditions (logic) in the method level. So, the conditions used in the subclass method has to be same as that of base class !?

(!!!) Can it be stronger?]

Open Closed Principle : choosing extension over change.

The classes should be open for extension but closed for change.

C L O S E Once the class is tested to meet the requirements and is stable it should be closed.
- should fix bugs if any
- this helps in avoiding any side effects

O P E N The class needs to be extended or built upon

① Inheritance

Extend the subclass behavior via inheritance.
The ~~sub~~ baseclass is not changed.

② If a class is an abstract/ interface [Polymorphism]
- through polymorphism
- by each concreate subclass providing its own version of implementation.

Features

→ helps in seperating out the stable parts of the design from varying parts.

→ Loose coupling

→

Dependancy Inversion

- Software dependancy or coupling

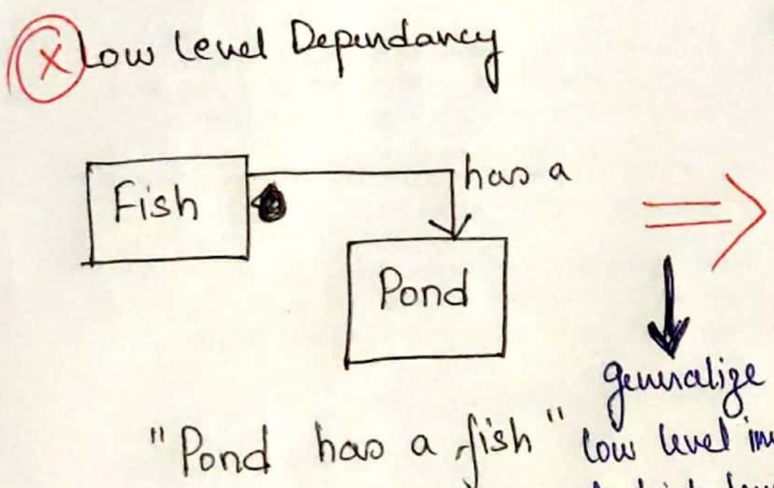
⊕ Robust ⊕ loosely coupled

⊕ Flexible

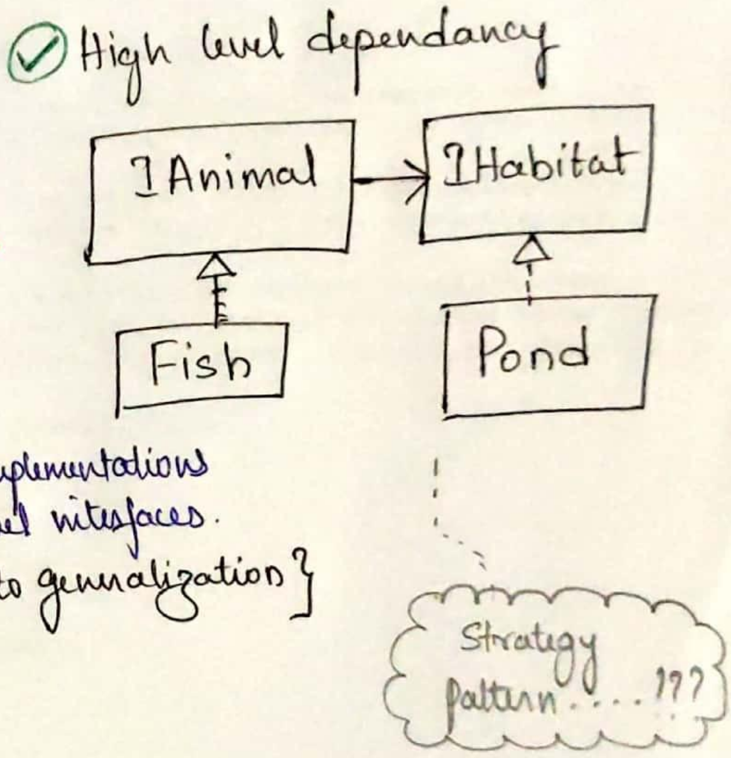
The principle states that high level modules should depend on high level generalizations, and not on low level details.

- client classes should dependent on interfaces & not the concreate classes

low level resource : concreate class (implementation)
high level resource : abstract or interfaces.
(provides general behaviors)



low level implementations to high level interfaces.
{indirection to generalization}



Composing Objects Principle

[All about code re-use]

loose Coupling
→ Generalization
→ Abstraction
→ Polymorphism

*to reduce tight coupling *gain code reuse

- One cause of tight coupling in our designs is inheritance
- Although inheritance provides great reusability, what it lacks is ~~to~~ provide loose coupling.
- We use this principle to avoid the tight coupling with inheritance.

(+) High code reuse without inheritance

This principle states that classes should achieve code reuse through (composition) aggregation rather than inheritance.

{ Aggregation and delegation offer less coupling than inheritance }

Advantages of composition over inheritance

1) The design can be extended or changed dynamically. ~~flexibility~~. at run time
2) Loosely coupled design by seperating the varying & non-varying parts.
3) Flexibility of being private! classes that work together without having to share any properties or behaviors.

Here, by code reuse, it does not ~~mean code to~~ imply to code duplication!
Code reuse is simply, to be able to effectively use the existing code in our designs.
Two ways are: {detailed ex in next page}

Disadvantage of composition

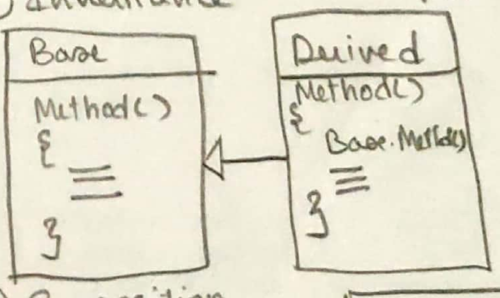
1) Similar implementations ~~across classes~~. across classes. as we do not have the code share (benefit of inheritance)

~~Code Share : Inheritance :: Code Reuse~~

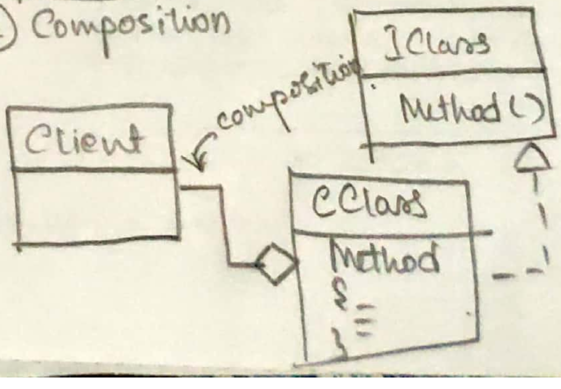
Codeshare : Inheritance :: Code reuse : Composition

- Provides code reuse without tight coupling
- Dynamically add behaviors @ run time
- More flexibility.

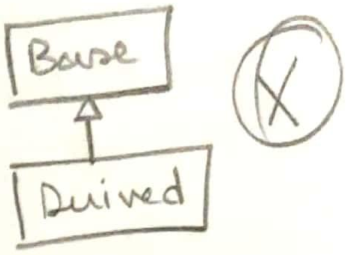
[X] ① Inheritance



[✓] ② Composition



Misusing ~~code~~ inheritance for code reuse
[Do NOT DO THIS! LIKE EVER!!!]



```
Class Base
{
virtual
 void Somefunc()
 {
   //do something
 }
 ---
}
```

```
class Derived : Base
{
 void Somefunc()
 {
   Base.Somefunc()
   //do something more
 }
```

$\longrightarrow$ Here we want to ~~[illegible]~~ extend the behavior of 'Somefunc' by inheriting in derived class.

The above should not be done for
1) Tight coupling b/w Base & Derived
2) Also this ~~[illegible]~~ violates the Liscov's Substitution principle

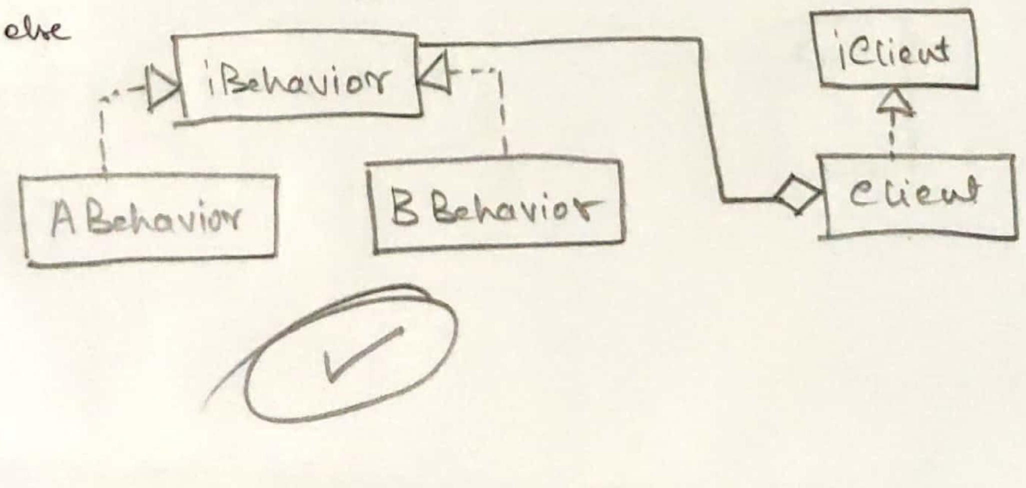
Instead do the following to fix the problem with inheritance.

SOLUTION : COMPOSITION

```
interface iBehavior
{
  void Somefunc();
}

Class ABehavior : iBehavior
{
  void Somefunc()
  {
    // Something
  }
}
class BBehavior : iBehavior
{
  void Somefunc()
  {
    // something else
  }
}
```

```
Class Client : iClient
{ iBehavior toDo;
  void DoAction()
  {
    // something
    toDo = new ABehavior;
    toDo.Somefunc();
    OR
    toDo = new BBehavior;
    toDo.Somefunc();
    // something;
  }
}
```



Interface Segregation Principle

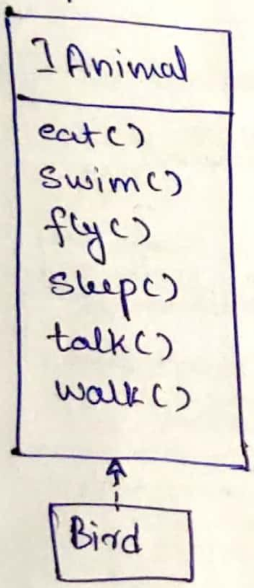
Solution to the dummy implementation problem.

- Any concreate class that implements an interface should not have any dummy implementations for any of the methods that are defined in the interface.
- This principle states that a class should not be forced to depend on methods it does not use.

SOLUTION

To split or segregate large interfaces into smaller generalizations.

Example :



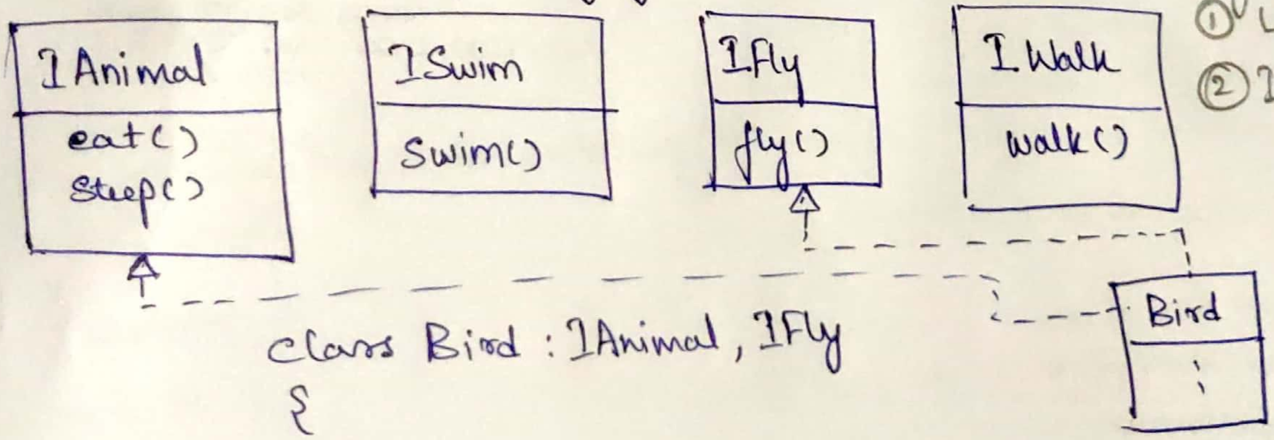
If all animal species were implemented from iAnimal, it would lead to dummy methods in specific animal species.

```
class Bird : IAnimal
{
   void fly()
   {
     Print(I am flying)
   }
   void swim()  } dummy
   {            }  code
     return;    }
   }
}
```

isn't similar to what LSP says!?

By doing so, we violate 2 design principles
① LSP
② ISP

Applying the interface segregation :



```
class Bird : IAnimal, IFly
{
   ⋮
}
```

Principle of Least Knowledge

Mind your ☐ business! (Law of demeter)

- how to manage complexity?

(+) Reduces coupling

(+) Provides stability.

Rules : How to know if you are really minding your own business? These rules provides a way to check if we are violating this principle.

1] A method 'M' ~~can only call~~ of class 'C' can only call other methods of the same class 'C'.

I am talking to myself, so... I must be minding my own business!

2] The method 'M' if takes a parameter 'P', can now call methods of the parameter 'P'.

My friend just told me she is not well, I can may be ask how she is feeling now. That won't make me too intrusive!

3] The method 'M' if it has a local variable within it of class 'C', can call on the methods of class 'C'.

Wow! I just wrote this awesome blog. Let me read it again & review it. I am still minding my own business, you see?

4] If class 'C' has a method 'M' and it also ~~has~~ has an instance of class 'P', the method 'M' can call upon the methods of class 'P'.

My parents bought this neat PS4! I am playing it now & for sure minding my own business! (unless my bro wants to play it too ... I may have to fight him :P)

The following would be violating the Law of demeter

I asked my mom to ask my brother to play less & study more.... I definitely am not minding my own business.

~~M~~

```
void M()
{
    ~~class~~ C = new C();
        C.somefunc();
        C.I.X.Somefunc(); ✗
        C.I.Somefunc(); ✗
}
```

↳ reach though → Chaining method Calls.

5] If the method call ~~from~~ returns a type to you that you are not aware of, do not try to use it.

This parcel is not addressed to us, maybe I should not take it!
I am minding my own business.

6] Classes should know as little as required.

Ignorance is bliss!

Other Principles of OOAD

Are these liquids or gas....? Coz they sure are NOT SOLID :D :D
PunIntended

1] Dont Repeat Yourself (DRY)
- Dont duplicate code by reuse it....
[III^r to dependency inversion & composing objects...?]

How to avoid code duplication? TRIVIA TIME!
Code reuse
How do you "code reuse"?
via generalization, abstraction, interfaces ...duh!

⊕ Maintenance

2] Encapsulate what changes [EWC, now why is 'EWC' not fun!
why only 'DRY'!]

3] Favor composition over inheritance.

4] Program to interface & not implementation.

5] Delegation Principle
[Goes on same lines of single responsibility principle]
Dont do all the things in one class ~~delegate~~ split the tasks into managable behaviors & delegate.