S.O.L.I.D PRINCIPLES

Created by – Arnab Chakraborty,

Consultant Technology at AchieveX Solutions Private Limited.

The entire documentation has been created to enhance the understanding of the principles behind writing clean, readable and functional programming codes that will be served purposefully.

Prerequisites:

1. Knowledge on Java (for syntactical understanding)
2. Understanding of Object-Oriented Programming (OOP) concepts

The concepts that are going to be discussed will eventually build up the understanding of some of the best code writing practices and principles that can be applicable into a broad range of different software programming languages.

Through this documentation you will be introduced to each and every **SOLID principle** in the scope of a project - **Design a Zoo using OOP**

Zoo

**Parent Class**

**Child Classes**

Authority

Animals

**Child (sibling) Classes**

**Different Zoo Functionalities**

**Different Species of Animals**

Monitoring

Maintenance

**Different attributes and characteristics**

Implementation of OOPs and SOLID Principles

The implementation of the principles will be done sequentially and to understand the effectiveness, the existing code will be measured in four main dimensions –

* Readability
* Testability
* Extensibility
* Maintainability

In terms of implementing the design of the ‘Zoo’ class we are at first going to design the ‘**Animal**’ class.

// Define the class - ‘**Animal**’

class Animal {

*// attributes [properties]*

    String species;

    double weight;

    String colour;

    int numberOfLegs;

    boolean hasWings;

    boolean hasFins;

*// behaviour [methods] (defined later)*

    void eat() {...}

    void swim() {...}

    void fly() {...}

    void breatheUnderWater() {...}

    void breatheOnLand() {...}

}

// Expand (define) the method swim() –

void swim() {

*// decide based on the species:*

*if*(species == "Tuna" || species == "Shark")

            print("Can swim..");

*else* *if*(species == "Pigeon" || species == "Tiger")

            print("The selected species can't swim at all !!")

    }

**Readable?**

* It is readable at this particular time.

What is the reason that the code is not readable?

* Q. Imagine that there are thousands of species. Do you think that reading that code would be easy in that case?
* Ans. If I have lots of species and complex logic, then I will have to read thousands of lines of code to understand how one specific animal swims. That is why the code is NOT READABLE.

**Testable?**

* It seems like it is testable.

void swim() {

*// decide based on the species :*

        String prompt =  "Can swim fast..";

*if*(species == "Tuna" || species == "Shark")

            print("Can swim.." + species + " " + prompt);

*else* *if*(species == "Pigeon" || species == "Tiger")

            print("The selected species can't swim at all !!")

*else* *if*(species == "Frog")

        print("I'm an amphibian, so I kinda swim !!");

*// --- --- --- ---*

*if*(species.beginsWith("T")) {

            ...

        }

    }

* Q. After adding a lots of species and complex logics, does the code still remain testable?
* Ans. Absolutely not! It’ll be a hectic job for the testers.

**Extensible?**

* Yes, we can simply add another species just by putting another else-if condition to handle a new species :

void swim() {

*// decide based on the species :*

*if*(species == "Tuna" || species == "Shark")

            print("Can swim..");

*else* *if*(species == "Pigeon" || species == "Tiger")

            print("The selected species can't swim at all !!")

*else* *if*(species == "Frog")

        print("I'm an amphibian, so I kinda swim !!");

}

**Maintainable?**

* It seems maintainable if multiple developers are working on this, suppose Dev1 is working on the species “Tuna”, Dev2 is working on the species “Shark”, Dev3 is working on the species “Pigeon” and so on.

But ultimately this particular code is neither Readable and Testable nor Extensible and Maintainable.

**Requirement :**

**Now, after that ‘prompt’ implementation there is another requirement for the species == “shark” that it can attack humans. Therefore, we just need to add another alert message with the particular description output.**

String prompt = "Can swim fast..";

String alertMessage = "Can attack humans!!";

if (species == "Tuna" || species == "Shark")

print ("Can swim..." + species + " " + prompt + alertMessage);

**Testable?**

* Q. Is this still testable?
* Ans. It’ll entirely break the test-cases, because as the ‘alertMessage’ is getting included for ‘sharks’ into the condition, it’ll also be actionable for the ‘Tuna’ as well.

So definitely the code for “Tuna” gets affected and broke as well.

Modifying the behaviour of one species can affect the behaviour for all other species.

**Maintainability?**

* If different developers are working on different species, to modify or add more behaviour and functionalities to species and try to submit their code (git commit),

**what will happen?**

Ans. **Merge Conflict.**

How to fix this?

* Single Responsibility Principle

Before adding another method / alertMessage in the code everything works fine in the QA, then you add one message and suddenly the code breaks –

Modifying the behaviour of one species can affect the behaviour for all other species.

Multiple developers commit making code collision -

Merge Conflict.

**S - Single Responsibility Principle comes into the place**

**Segregate the responsibilities.**

* Every function / class / module / unit-of-code should have a single and well-defined responsibility.

(Any piece of code should have only one reason to change.)

* **If some class is serving multiple responsibilities**, we should split it into multiple classes.

**Solution:**

**Effective utilization of the OOP concept -> sub class**

class Tuna extends Animal {

    @override

    void swim() {

        print("Swim fishy swim but do not attack humans !!");

    }

}

class Shark extends Animal {

    @override

    void swim() {

        print ("....");

    }

}

Split the ‘Animal’ class into multiple sub classes and override the swim() method.

**Readable?**

* No, not yet. There is going to be a lot of classes for different species now.
* It might seem bad, but it’s not too bad.
* **Readability** does not mean less code. It means that each piece of code is individually readable.

**Explanation** -> You don’t know what all the thousands of species are and their behaviour, but if you’re particularly needed to work on the species “Tuna”, then you can simply go for the “Tuna.java” file / class and can get the entire details on Tuna.

Q. So now, when you’re looking at the particular class “Tuna”, does this seem readable or not?

class Tuna extends Animal {

    @override

    void swim() {

        print("Swim fishy swim but do not attack humans !!");

    }

void anotherMethod() {…}

void anotherMethod() {…}

}

Ans. Yes. Extremely readable, because each class is small and to-the-point.

To understand Tuna, you just don’t need to go for reading the entire ‘Animal’ class.

**Testable?**

* Any changes to ‘Tuna.swim()’ will not affect ‘Shark.swim()’. Test cases are separated – they are not coupled.

The code is still not perfect, but it’s direction is now at least towards the right way.

Extensible?

* Can I add a new species?

Ans. Yes, obviously. Just by adding a new class.

Maintainable?

* Did we get rid of the merge conflicts to some extent?

Ans. Yes. Step in the right direction.

That is what SOLID Principles are. They will help your code to strengthen it from all different points.

------------------- End of part-1 ------------------

**Requirement :**

**There is a new requirement to create (design) a bird ->**

*// Create a class for "Bird" that is also an Animal:*

class Bird extends Animal {

*// inherits the string 'species' from the 'Animal' class*

    void fly() {...}

}

*// Configure the 'Bird' class for all the different bird species available in the zoo.*

class Bird extends Animal {

*// same if-else ladder to decide how the bird flies ->*

    if(*species* == "Sparrow")

        print("Fly low");

    else if(*species* == "Eagle")

        print("Glide elegantly");

    else if(*species* == "Albatross")

        print("Fly with Physics hack !!");

}

**Neither readable & testable nor extensible & maintainable.**

**Readable?**

**Testable?**

**Extensible?**

* On the surface it seems extensible, because all we need to do, is to add a new else-if condition.

But the code is not exactly extensible.

**In a scope of a project - to construct our entire code, we sometimes need to call libraries and import packages.**

* Actually, the code is not extensible. Because, the code is not editable / permissible to be edited.

Now, one might ask question like – if I’m importing a library that is not extensible, is not actually my fault. But the thing is that – sometimes you’ll also be that person.

* Open Close Principle

**O – Open-Close Principle comes into the place**

**Code should be closed for modification, yet still, open for extension!**

**Solution:**

**Let code close to modification, but open for extension.**

**Why to use Open-Close principle?**

* Developers – write code. Test the code on their system.
* QA team – test, integration tests (can even take months)
* Deployment Phase:
  + Staging server – tested internally
  + A/B testing
    - Shipped out to only 5% of the userbase
    - Monitor the errors / performance during this period
  + Finally, it will be deployed to 100% userbase

**Modifying code for a small application can be done easily, but if you just think about a large application (Google / Microsoft’s project such as modifying the code of ‘Google Maps’ or ‘Microsoft Office’), it’ll be very much hectic and may even lead towards crashing the entire application.**

**Therefore, modifying code repetitively is a bad habit for developers.**

*// Instead of having the above code pattern (Library + Client) :*

*// Library :*

[library] Open-Close\_Principle\_Zoo

class Sparrow extends Bird {

    @override

    void fly() {

        print("I love to fly around..");

    }

}

*// Client code :*

[executable] Client {

    import Open\_Close\_Principle\_Zoo;

    class main() {

        void main {

            class Peacock extends Bird {

                @override

                void fly() {

                    print("Pe-hens (female) can fly, but males can not !!");

                }

            }

        }

    }

}

**Extensible?**

* Now open to extension.
* Still not open to modification.

Client doesn’t have access to the library code, but still can access it and extend it.

------------------- End of part-2 ------------------

Q. Didn’t we make the exact same change for the Single Responsibility Principle (SRP) as well?

Ans. Exactly yes. We took the ‘Animal’ class and we split it down into different different species.

Q. Is the SRP the same as Open/Close?

Ans. No! The change we did was same, but the intent was different.

**Earlier on the intent was to ensure that the class has only one single responsibility.**

**Now, the intent is to make the code – open to extension, but to close for modification.**

**This implies that the SOLID principles are linked to each other.**

**Requirement :**

**Just think of a scenario where there is a requirement to fetch a list of bird details that can’t fly ->**

Now, we have another bird named ‘Kiwi’ that can’t fly.

Define the ‘Kiwi’ class as an abstract class.

Then, how to implement the Kiwi.fly() method?

* Not implementing the ‘fly’ method will cause an error.
* Throwing an exception will also cause an error. Why?

void fly() {

*throw* *new* nonFlyingBirdException(String species) {

                 (species + " don't fly");

}

}

The answer of why: The executable client code contains the fly() method. But as we’re not referencing the method inside the ‘Kiwi’ class, it’ll generate an error in the client’s executable code.

// Client’s executable code :

[executable] Client {

            class Main() {

                void main() {

                    Bird bird;

                    bird = getBirdFromUserChoice(); *// might be any 'Bird' subclass*

                    bird.fly();

                }

            }

        }

**Before extension:** Everything just works fine.

**After extension:** As the ‘Kiwi’class doesn’t implement the ‘fly()’ method, Client’s code will throw-out an error.

Just realize: We didn’t make any change (modify) neither into the client’s code nor into the library code.

All the classes are exactly same; however, code now breaks!

------------------- End of part-2 ------------------

**Solution:**

The way to solve this problem is –

* Liskov Substitution Principle

**L-Liskov Substitution Principle**

* Any functionality in the parent class must also work for child classes.
* If some piece of code works with a parent class ‘P’, then the code must also work, without modifications, with any child class, such as – ‘class C extends P’
* Result: Any extension to existing code should not break existing features.

How should we redesign the above scenario to follow the LSP?

*// => Liskov Substitution principle :*

    abstract class Bird extends Animal {

        String Species

*// Bird attributes -*

        int numberOfWings;

        boolean hasBeak;

*// Bird behaviours -*

        void speak;

*// no fly() method is here, because we know that 'all birds can't fly'*

    }

*// segregate the behaviours (functionalities) using interface*

    interface ICanFly {

        void fly();

    }

    class Eagle extends Bird implements ICanFly {

        @Override

        void fly() {

            print("Glide elegantly");

        }

    }

    class Kiwi extends Bird {

*// We don't have to provide the fly() method by implementing the 'ICanFly' interface , because 'Kiwi' can't fly*

------------------- End of part-3------------------

* Interface Segregation Principle

**I – Interface Segregation Principle**

- Keep your interfaces minimal

- No code should be forced to implement a method that it does not read

The problem that has appeared is –

Implementing Interface has already solved that problem of implementing the ‘fly()’ method into different Bird species.

But in a scope of a large project (maintaining the entire zoo, there is a requirement of drones. As a ‘Drone’ can also fly, if it just incorporates that method – errors will be occurring while implementing the ‘flapWings()’ or ‘kickToTakeOff()’ method.

Isn’t this simply the SRP applied to interfaces?

* All the SOLID Principles are linked.

Also consider another scenario related to that to understand ‘Interface Segregation Principle’ better –

Designing a cage:

*// => Design a cage :*

    interface IBowl {...}

    class MeatBowl implements IBowl {...}

    class FruitBowl implements IBowl {...}

    interface IDoor {...}

    class IronDoor implements IDoor {...}

    class WoodenDoor implements IDoor {...}

*// for Tigers -> '*

    class Cage1 {

        MeatBowl bowl;

        WoodenDoor door;

        List<Tiger> tigers;

        public Cage1 {

*// add 5 tigers inside :*

*for*(int i = 0; i < 5; i++)

                tigers.add(*new* Tiger(...));

        }

    }

    class Cage2 {

        FruitBowl bowl;

        WoodenDoor door;

        List<Sparrow> sparrows;

        public Cage2 () {...}

    }

How many different types of closures / cages can be created?

* Several different cages for different species.

But the thing is that all the cages are going to be formed like that, that means the attributes for the Cages are going to be almost similar.

What is wrong in this code?

* Too many cage classes.
* Repetitive code. (Code duplication)

*// for Tigers -> '*

    class Cage1 {

        MeatBowl bowl;

        WoodenDoor door;

        List<Tiger> tigers;

        public Cage1 {

*// add 5 tigers inside :*

*for*(int i = 0; i < 5; i++)

                tigers.add(*new* Tiger(...));

        }

    }

**This code is exactly similar to**

class Cage2 {

        FruitBowl bowl;

        WoodenDoor door;

        List<Sparrow> sparrows;

        public Cage2 () {...}

    }

**There is a concept in programming – DRY : Do Not Repeat Yourself!**

Cage1 class depends on the ‘MeatBowl’, ‘Tiger’ and ‘IronDoor’

‘MeatBowl’ depends on the interface ‘IBowl’,

‘Tiger’ depends on the interface ‘IAnimal’ and

‘IronDoor’ depends on the interface ‘IDoor’

IBowl IAnimal IDoor - High Level Abstraction

MeatBowl Tiger IronDoor - Low Level Details (Specific type, e.g. a tiger is a .. very specific type of animal)

Cage1 - High Level (Composite) class (Connects multiple /……………… functionalities together)

The issues that we have here are –

* High level class ‘Cage1’ depends on the concrete, low-level types ‘MeatBowl’, ‘Tiger’ and ‘IronDoors’.

------------------- End of part-4------------------

* Dependency Inversion Principle

**Solution:**

**D- Dependency Inversion Principle**

* High level modules should NOT depend on low-level modules
* Instead, they should depend on high-level Abstraction (basically interfaces and abstract classes)

How Dependency Inversion Principle comes to the solution?

Ans. By eliminating the middle level (dependency on the low-level details)

The Cage will directly be dependent upon the High Level Abstraction like – Ibowl, IAnimal, IDoor and so on.

Now, how to implement the Dependency Inversion Principle?

Implementing Dependency Injection

The principle is Dependency Inversion, but the way we’ll achieve this is – ‘Dependency Injection’.

Instead of creating the dependencies, we’ll inject them.

Redesign the class –

*// Instead of using the below code :*

    interface IBowl;

    class MeatBowl implements IBowl {...}

    class FruitBowl implements IBowl {...}

    interface Door{...}

    class IronDoor implements Door{...}

    class WoodenDoor implements Door{...}

*// implement this :*

    class Cage {

        IBowl bowl;

        IDoor door;

        List<Animal> animals;

*// constructor :*

        public Cage(IBowl *bowl*, IDoor *door*, List<Animal> . . *habitants*) {

            this.bowl = bowl;

            this.door = door;

            this.animals.addAll(habitats);

        }

    }

The advantage of the code :

Assume that the below code is an executable client code where we're implementing this cages for different species using Dependency Inversion :

class Main {

        void main() {

*// creating a cage for tigers :*

            Cage cage1 = *new* Cage (*new* MeatBowl(),

*new* Irondoor(),

                                   Arrays.toList(*new* . . . . . . . Tiger(T1),(T2)))

        }

*// in the same manner - create a cage for sparrows :*

           Cage cage2 = *new* Cage (*new* FruitBowl(),

*new* Woodendoor(),

                                  Arrays.toList(*new* ….. . . Sparrow(SP1),(SP2)))

        }

    }

We’re having the same class –

class Cage {

        IBowl bowl;

        IDoor door;

        List<Animal> animals;

*// constructor :*

        public Cage(IBowl *bowl*, IDoor *door*, List<Animal> .. . . . . . *habitants*) {

            this.bowl = bowl;

            this.door = door;

            this.animals.addAll(habitats);

        }

    }

no code repetition, but now we can use this class as much number of possible ways we want.

How? By injecting the dependencies.

We inverted the thing.

Instead of having fixed set of dependencies, we are now creating now dependencies and injecting them.

Previously the dependencies were fixed – IronDoor, MeatBowl.

But now we are creating the dependencies by our own as per the requirement.

Now, the High Level Module ‘Cage’ now depends only on the High-Level abstractions.

---------------------- The End --------------------