



Object Oriented Design Principles

SOLID

(IT069IU)

Le Duy Tan, Ph.D.



ldtan@hcmiu.edu.vn



leduytanit.com

Previously,

- Generic
 - Generic Class
 - Bounded type parameters
 - Generic Method

Agenda



- Object Oriented Design Principles: SOLID
 - S: Single responsibility
 - O: Open/closed principle
 - L: Liskov substitution principle
 - I: Interface segregation principle
 - D: Dependency inversion principle



So far,

We have learnt about the **four main OOP principles**:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Encapsulation

When an object only exposes the selected information.

Abstraction

Hides complex details to reduce complexity.

Inheritance

Entities can inherit attributes from other entities.

Polymorphism

Entities can have more than one form.



Object Oriented Design (OOD) Principles

Golden rules used by object-oriented developers since the early 2000s!

Intro

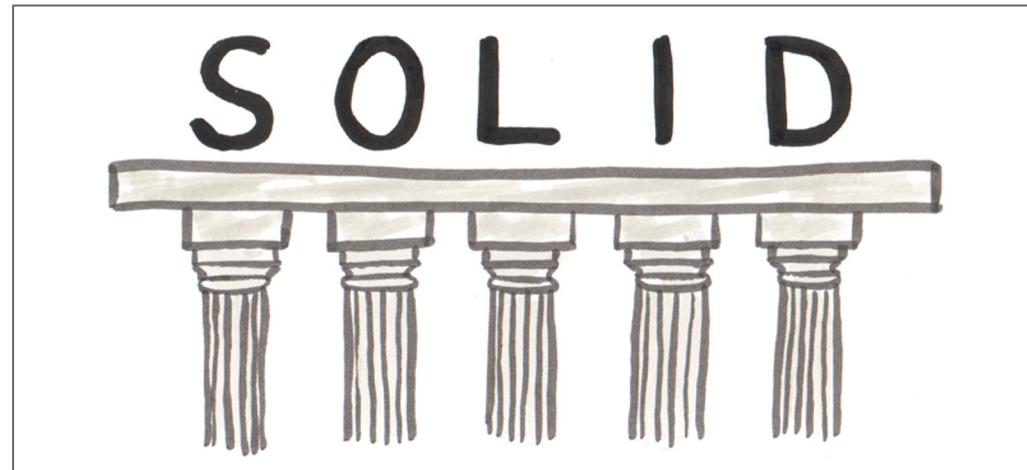
SOLID is an acronym for the first **five object-oriented design (OOD) principles** by Robert Martin (also known as Uncle Bob). They are a set of rules and best practices to follow while designing a class structure.

These principles establish the best practices to create more **maintainable**, **understandable**, and **flexible software**. As our applications grow in size, we can reduce their complexity and save ourselves a lot of headaches.

They all serve the same purpose: "To create **understandable**, **readable**, and **testable code** that many developers can collaboratively work on."

SOLID stands for:

- **S** - Single-Responsibility Principle
- **O** - Open-Closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle



S - Single-Responsibility Principle (SRP)



Single-responsibility Principle (SRP) states:

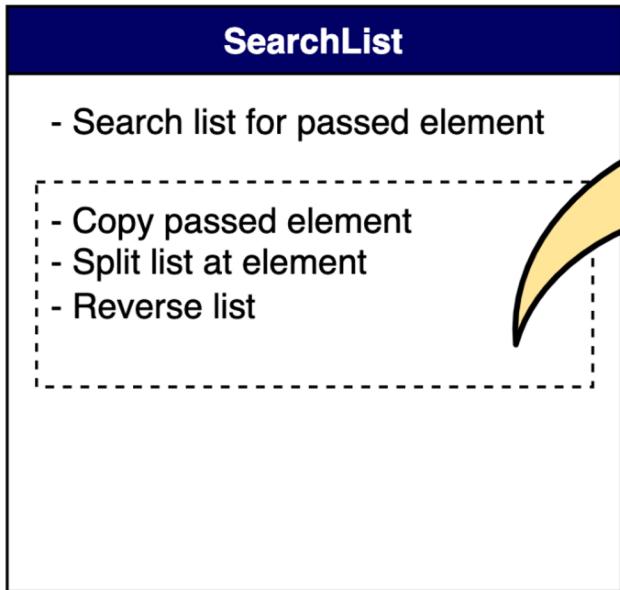
"A class should have only one reason to change, meaning that a class should have only one job."

Explanation:

- A class should have full responsibility for **a single functionality** of the program. The class should contain only variables and methods relevant to its functionality -> **fewer bugs and less dependencies**.
- **Smaller, well-organized classes are easier to manage** than monolithic (big) ones.



S - Single-Responsibility Principle (SRP)



SRP: Splitting a non-SRP class

SRP Examples 1

- Imagine that employees of a software company need to do 1 of 3 things:
 - software programmer (developer),
 - software tester (tester),
 - software salesman (salesman).
- Each employee will have a title and based on the title will do the corresponding job.
- Question: Then should you design class “Employee” with property “position” and 3 methods developSoftware(), testSoftware() and saleSoftware()?

```
class Employee
{
    string position;

    void developSoftware() {};
    void testSoftware() {};
    void saleSoftware() {};
}
```

SRP Examples 2



- **The answer is NO.**
- Imagine if there was one more position, human resource manager, we would have to modify the "Employee" class, add a new method?
- What if there were 10 more positions?
- At that time, the created objects will have a lot of redundant methods: Developer does not need to use testSoftware() and saleSoftware() functions right, accidentally using the wrong method will also have unpredictable consequences.
- **Solution with the principle of Single Responsibility:** 1 responsibility per class. We will create an abstract class "Employee" whose method is working(), from here you inherit 3 classes namely Developer, Tester and Salesman.
 - In each of these classes, you will implement a specific working() method, depending on the task of each person.

SRP Examples 2



The program above does not follow SRP because RegisterUser does three different jobs: register a user, connect to the database, and send an email.

```
// does not follow SRP
public class RegisterService
{
    public void RegisterUser(string username)
    {
        if (username == "admin")
            throw new InvalidOperationException();

        SqlConnection connection = new SqlConnection();
        connection.Open();
        SqlCommand command = new SqlCommand("INSERT INTO [...]"); //Insert user into database.

        SmtpClient client = new SmtpClient("smtp.myhost.com");
        client.Send(new MailMessage()); //Send a welcome email.
    }
}
```

Problem: This type of class would cause confusion in larger projects, as it is unexpected to have email generation in the same class as the registration.

There are also many things that could cause this code to change like if we make a switch in a database schema or if we adopt a new email API to send emails.

SRP Solution 2



Instead, we need to split the class into three specific classes that each accomplish a single job. Here's what our same class would look like with all other jobs refactored to separate classes:

```
public class RegisterService
{
    public void RegisterUser(string username)
    {
        if (username == "admin")
            throw new InvalidOperationException();

        userDatabase.Insert(...);

        emailService.Send(...);
    }
}
```

This achieves the SRP because **RegisterUser** only registers a user and the only reason it would change is if more username restrictions are added. All other behavior is maintained in the program but is now achieved with calls to **userDatabase** and **emailService**.

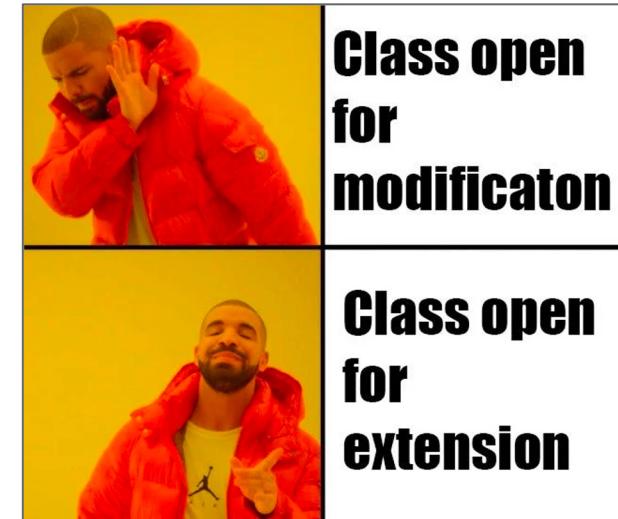
0 - Open-Closed Principle (OCP)

Open-Closed Principle (OCP) states:

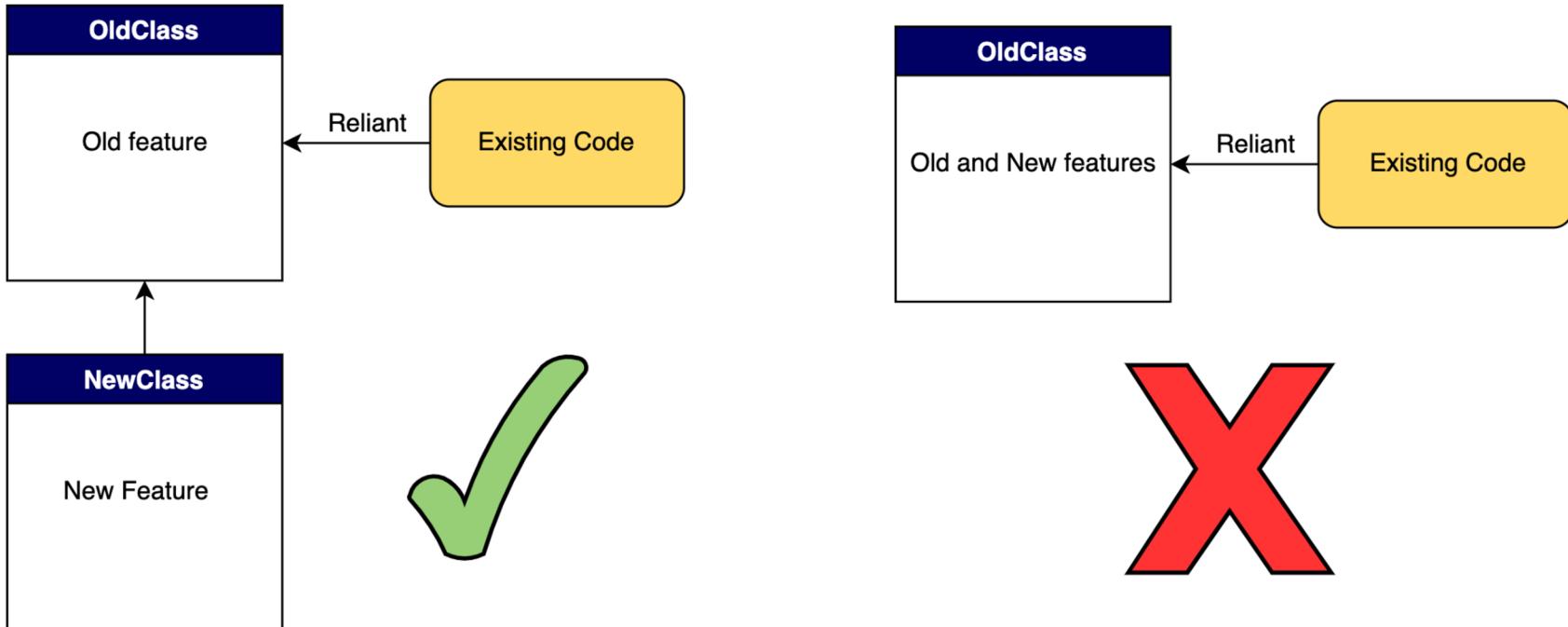
“A class should be open for extension and closed to modification.”

Explanation:

- Modification means changing the code of an existing class, and extension means adding new functionality.
- We should be able to **add new functionality without touching the existing code for the class**. This is because whenever we modify the existing code, we are taking the risk of creating potential bugs. So we should avoid touching the tested and reliable production code if possible.
- But how are we going to add new functionality without touching the class, you may ask. **It is usually done with the help of interfaces and abstract classes.**



O - Open-Closed Principle (OCP)



OCP: Modifying OldClass instead of extending

OCP Example 1



We need a class to handle the connection to the database. The original design only included SQL Server and MySQL. The original design looks like this:

```
class ConnectionManager
{
    public void doConnection(Object $connection)
    {
        if ($connection instanceof SqlServer) {
            //connect with SqlServer
        } elseif ($connection instanceof MySql) {
            //connect with MySql
        }
    }
}
```

Problem: Then it is required to connect more to MongoDB and some other database systems. To add functionality we must add other else if blocks to the code, which makes the code cumbersome and harder to manage!

OCP Solution 1



- Apply Abstract to redesign classes of SqlServer, MySql, MongoDB...
- These classes all have the same task of creating a connection to the respective database, which can be called Connection.
- The way to connect to the database varies depending on the type of connection, but it can be called **doConnect**.
- So we have a base abstract class Connection that has a doConnect method, specific classes are SqlServer, MySql, MongoDB... inheriting from Connection and overwriting the doConnect method in accordance with that class.
- With this design, when you need to connect to a new type of database, just add a new class that inherits Connection without modifying the code of the ConnectionManager class, this satisfies 2 conditions of the OCP principle.

```
abstract class Connection()
{
    public abstract void doConnect();
}

class SqlServer extends Connection
{
    public void doConnect()
    {
        //connect with SqlServer
    }
}

class MySql extends Connection
{
    public void doConnect()
    {
        //connect with MySql
    }
}

class ConnectionManager
{
    public void doConnection(Connection $connection)
    {
        //something
        //.....
        //connection
        $connection->doConnect();
    }
}
```

OCP Example 2



In our book store, we want books to be saved to a database so that we can search them easily. We create the database, connect to it, and we add a save method to our Book class:

```
public class BookStore {
    Book book;

    public BookStore(Book book) {
        this.book = book;
    }

    public void saveToFile(String filename) {
        // Creates a file with given name and writes the book
    }

    public void saveToDatabase() {
        // Saves the book to database
    }
}
```

Problem: Unfortunately we, as the lazy developer for the book store, **did not design the classes to be easily extendable in the future. So in order to add this feature, we have modified the BookStore class.** If our class design obeyed the Open-Closed principle we would not need to change this class.

OCP Solution 2



So, as the lazy but clever developer for the book store, we see the design problem and decide to refactor the code to obey the principle.

```
interface BookStore {  
  
    public void save(Book book);  
  
}
```

We change the type of BookStore to Interface and add a save method. Each class will implement this save method.

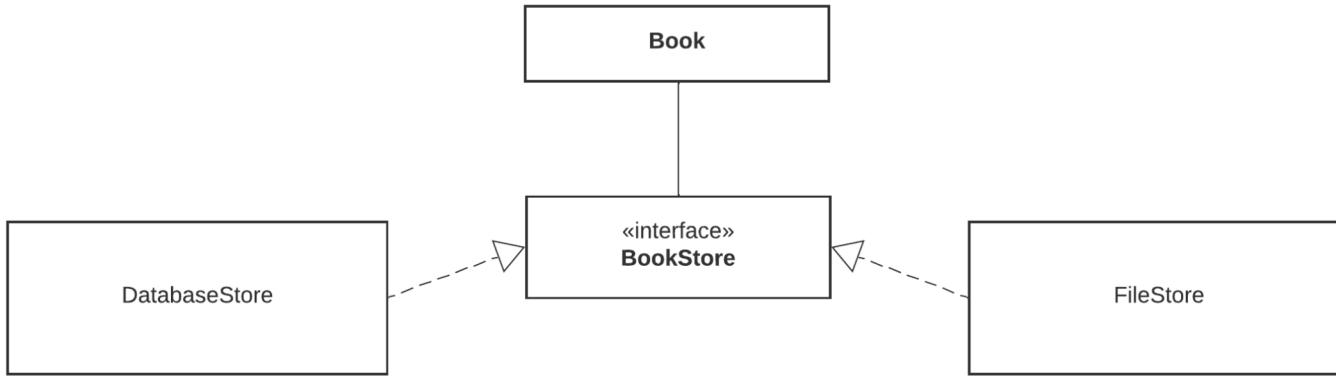
```
public class DatabaseStore implements BookStore {  
  
    @Override  
    public void save(Book book) {  
        // Save to DB  
    }  
  
}
```

```
public class FileStore implements BookStore {  
  
    @Override  
    public void save(Book book) {  
        // Save to file  
    }  
  
}
```

OCP Solution 2



So our class structure now looks like this:



Now our persistence logic is easily extendable. If our boss asks us to add another database and have 2 different types of databases like MySQL and MongoDB, we can easily do that.

You may think that we could just create multiple classes without an interface and add a save method to all of them.

We can now pass any class that implements the BookStore interface to this class with the help of polymorphism. This is the flexibility that interfaces provide.

OCP Example 3



The shipping logic of an order is placed inside the Order class.

```
public class Order {

    public long calculateShipping(ShippingMethod shippingMethod) {
        if (shippingMethod == GROUND) {
            // Calculate for ground shipping
        } else if (shippingMethod == AIR) {
            // Calculate for air shipping
        } else {
            // Default
        }
    }
}
```

Assuming the system needs to add a new shipping method, we have to add another case in the calculateShipping method. This will make the code very difficult to manage.

OCP Solution 3

Instead, we should decouple the shipping handling logic into a Shipping interface.
Interface Shipping will have many implementations for each form of transportation:
GroundShipping, AirShipping, ...



```
public interface Shipping {  
    long calculate();  
}  
  
public class GroundShipping implements Shipping {  
  
    @Override  
    public long calculate() {  
        // Calculate for ground shipping  
    }  
}  
  
public class AirShipping implements Shipping {  
  
    @Override  
    public long calculate() {  
        // Calculate for air shipping  
    }  
}  
  
public class Order {  
  
    private Shipping shipping;  
  
    public long calculateShipping() {  
        // Find relevant Shipping implementation then call calculate() method  
    }  
}
```

L: Lisko substitution principle (LSP)

Lisko substitution principle (LSP) states:

“Subclasses should be substitutable for their base classes.”

Explanation:

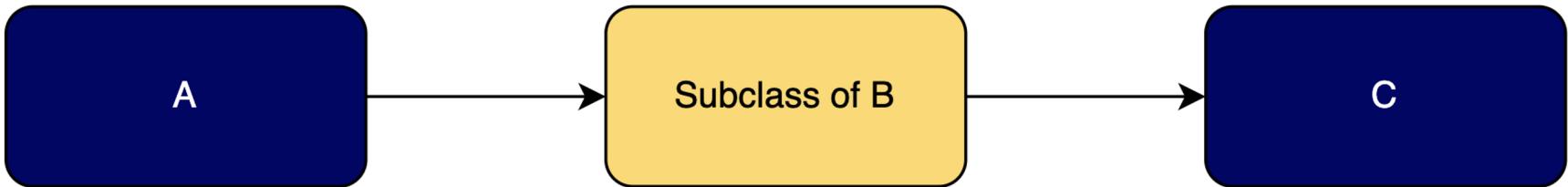
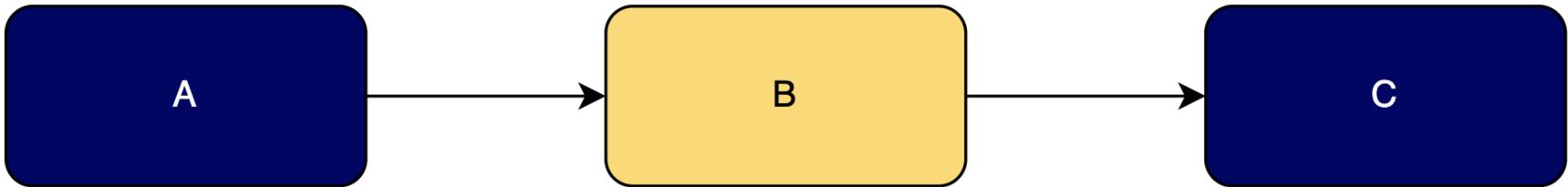
- This means that, given that class B is a subclass of class A, we should be able to pass an object of class B to any method that expects an object of class A and the method should not give any weird output in that case.
- This is the expected behavior, because when we use inheritance we assume that the child class inherits everything that the superclass has. The child class extends the behavior but never narrows it down.
- Therefore, when a class does not obey this principle, it leads to some nasty bugs that are hard to detect.
- Liskov's principle is easy to understand but hard to detect in code.



Liskov Substitution Principle

If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.

LSP Example



LSP: seamless substitution for the B class in existing code



LSP Example

We have the Animal interface and two implementations of Bird and Dog as follows:

```
public interface Animal {  
    void fly();  
}  
  
public class Bird implements Animal {  
  
    @Override  
    public void fly() {  
        // Flying...  
    }  
}  
  
public class Dog implements Animal {  
  
    @Override  
    public void fly() {  
        // Dog can't fly  
        throw new UnsupportedOperationException();  
    }  
}
```

It is clear that the Dog class violates the principle of Liskov substitution.

LSP Solution



The solution here would be: create a FlyableAnimal interface as follows:

```
public interface Animal {  
}  
  
public interface FlyableAnimal {  
  
    void fly();  
}  
  
public class Bird implements FlyableAnimal {  
  
    @Override  
    public void fly() {  
        // Flying...  
    }  
}  
  
public class Dog implements Animal {  
}
```

I: Interface Segregation Principle (ISP)



Interface Segregation Principle (ISP) states:

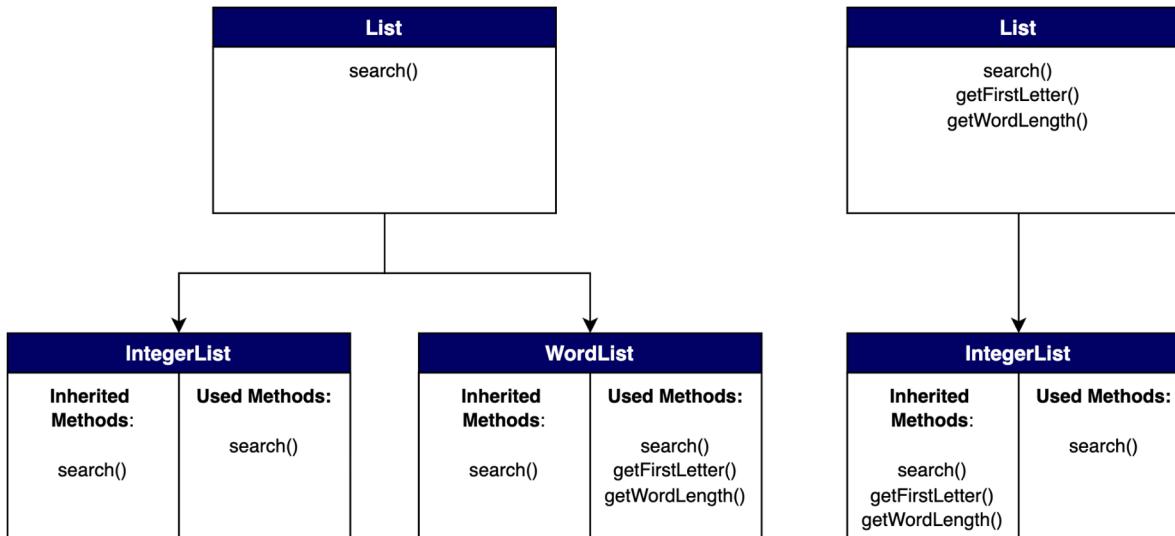
“Many client-specific interfaces are better than one general-purpose interface. A client should never be forced to implement an interface that it doesn’t use, or clients shouldn’t be forced to depend on methods they do not use.”

Explanation:



- Segregation means keeping things separated, and the Interface Segregation Principle is about **separating the interfaces**.
- **Larger interfaces should be split into smaller ones.** By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.
- This principle is easy to understand. Imagine we have a large interface, about 100 methods. The implementation will be very difficult because these interface classes will be forced to implement all the methods of the interface. There can also be redundancy because a class does not need to use all 100 methods. **When separating the interface into many small interfaces, including related methods, the implementation and management will be easier.**

I: Interface Segregation Principle (ISP)



ISP: Only inherit applicable methods



ISP Example 1



We have an Animal interface as follows:

```
interface Animal {  
  
    void eat();  
  
    void run();  
  
    void fly();  
  
}
```

We have two classes Dog and Snake that implement the Animal interface. But it's silly, how can Dog fly(), just like Snake can't run()?

ISP Solution 1



- Instead, we should split into 3 interfaces like this:

```
interface Animal {  
  
    void eat();  
  
}  
  
interface RunnableAnimal extends Animal  
{  
  
    void run();  
  
}  
  
interface FlyableAnimal extends Animal  
{  
  
    void fly();  
  
}
```

ISP Example 2



We modeled a very simplified parking lot. It is the type of parking lot where you pay an hourly fee.

```
public interface ParkingLot {  
  
    void parkCar();          // Decrease empty spot count by 1  
    void unparkCar(); // Increase empty spots by 1  
    void getCapacity();    // Returns car capacity  
    double calculateFee(Car car); // Returns the price based on number of  
hours  
    void doPayment(Car car);  
}  
  
class Car {  
    // Some implementation here...  
}
```

ISP Problem 2

Now consider that we want to implement a parking lot that is **free**.

```
public class FreeParking implements ParkingLot {  
  
    @Override  
    public void parkCar() {  
    }  
  
    @Override  
    public void unparkCar() {  
    }  
  
    @Override  
    public void getCapacity() {  
    }  
  
    @Override  
    public double calculateFee(Car car) {  
        return 0;  
    }  
  
    @Override  
    public void doPayment(Car car) {  
        throw new Exception("Parking  
lot is free");  
    }  
}
```

Our parking lot interface was composed of 2 things: **parking related logic** (park car, unpark car, get capacity) and **payment related logic**.

But it is **too specific**. Because of that, our **FreeParking class was forced to implement payment-related methods that are irrelevant**. Let's separate or segregate the interfaces.

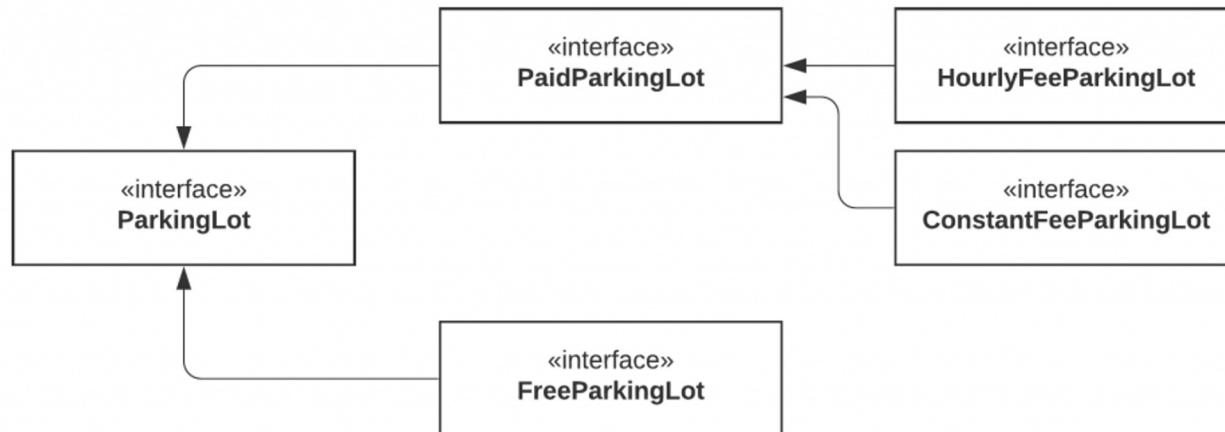


ISP Solution 2



We've now separated the parking lot. With this new model, we can even go further and split the PaidParkingLot to support different types of payment.

Now our model is much more flexible, extendable, and the clients do not need to implement any irrelevant logic because we provide only parking-related functionality in the parking lot interface.



D: Dependency Inversion Principle (DIP)



Dependency Inversion Principle (DIP) states:

"Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions."

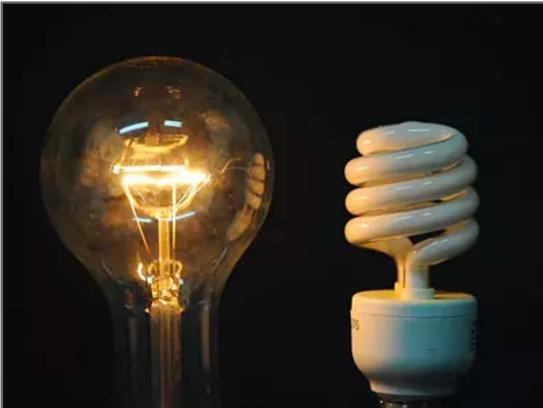
Explanation:

- Our classes should depend upon interfaces or abstract classes instead of concrete classes and functions.
- This principle is related to Open-Closed Principles (OCP).

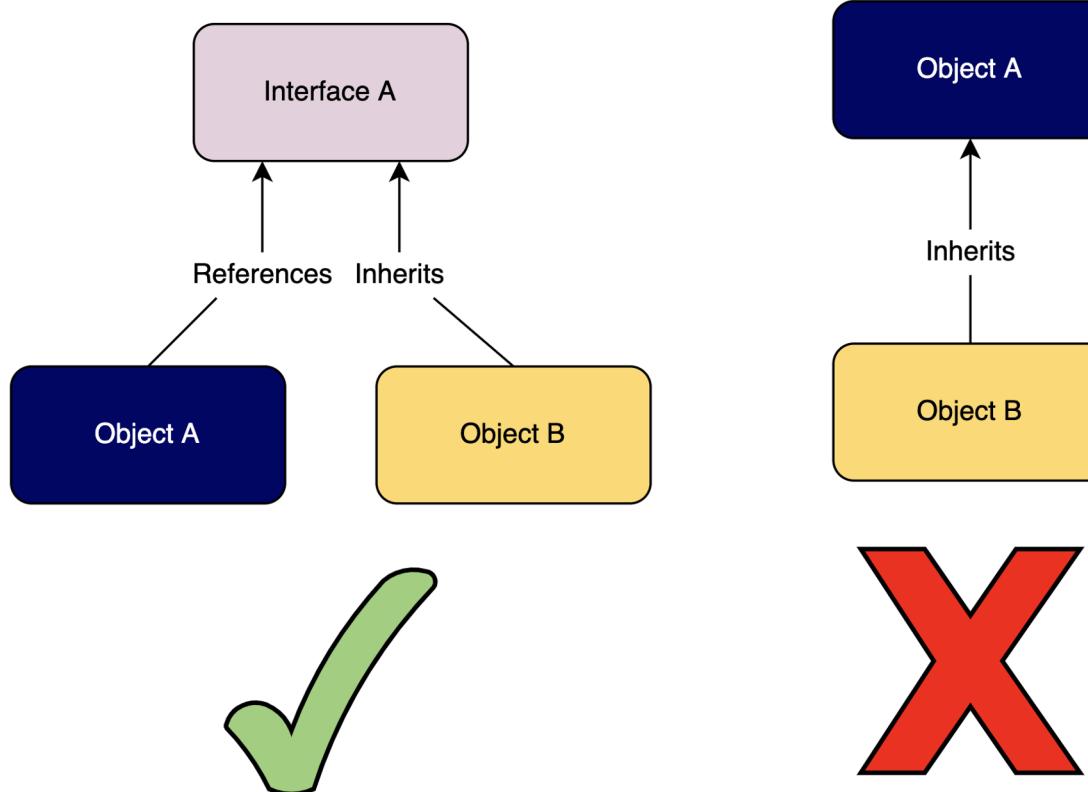


Dependency Inversion Principle

Would you solder a lamp directly to the electrical wiring in a wall?



D: Dependency Inversion Principle (DIP)



DIP: Changes to Objects A and B will not affect the other

DIP Example 1

To demonstrate this, let's bring to life a Windows computer with code:

```
public class WindowsComputer {}
```

But what good is a computer without a monitor and keyboard? Let's add one of each to our constructor so that every computer comes with a Monitor and a StandardKeyboard:

```
public class WindowsComputer {  
  
    private final StandardKeyboard keyboard;  
    private final Monitor monitor;  
  
    public WindowsComputer() {  
        monitor = new Monitor();  
        keyboard = new StandardKeyboard();  
    }  
  
}
```

This code will work, and we'll be able to use the StandardKeyboard and Monitor freely within our WindowsComputer class.

Problem solved? Not quite. By declaring the StandardKeyboard and Monitor with the new keyword, we've tightly coupled these three classes together.

Not only does this make our WindowsComputer hard to test, but we've also lost the ability to switch out our StandardKeyboard class with a different new one. And we're stuck with our Monitor class too.



DIP Solution 1

Let's decouple our machine from the StandardKeyboard by adding a more general Keyboard interface and using this in our class:

```
public interface Keyboard { }
```

```
public class WindowsComputer{

    private final Keyboard keyboard;
    private final Monitor monitor;

    public WindowsComputer(Keyboard keyboard, Monitor monitor) {
        this.keyboard = keyboard;
        this.monitor = monitor;
    }
}
```

Here, we're using the dependency injection pattern to facilitate adding the Keyboard dependency into the WindowsComputer class. Let's also modify our StandardKeyboard class to implement the Keyboard interface so that it's suitable for injecting into the WindowsComputer class:

```
public class StandardKeyboard implements Keyboard { }
```

Now our classes are decoupled and communicate through the Keyboard abstraction. If we want, we can easily switch out the type of keyboard in our machine with a different implementation of the



DIP Example 2



For example, we have 2 low-level **class BackendDeveloper** and **FrontendDeveloper** and 1 high-level **class Project** uses the above 2 classes:

```
public class BackendDeveloper {  
    private void codeJava() {}  
}  
  
public class FrontendDeveloper {  
    private void codeJS() {}  
}  
  
public class Project {  
    private BackendDeveloper backendDeveloper = new BackendDeveloper();  
    private FrontendDeveloper frontendDeveloper = new FrontendDeveloper();  
  
    public void build() {  
        backendDeveloper.codeJava();  
        frontendDeveloper.codeJS();  
    }  
}
```

Suppose if later, the project changes technology. Backend developers don't code Java anymore but switch to C code. Frontend developers don't code pure JS anymore but upgrade to JS frameworks like AngularJS. Obviously we have to fix not only the code in the low-level classes (**BackendDeveloper** and **FrontendDeveloper**) but also the code in the high-level classes (**Project**) that are using those low-level classes. This shows that high-level classes are having to depend on low-level classes.

DIP Solution 2

At this point, we will add a Developer abstraction to which the above modules depend:

```
public interface Developer {
    void develop();
}

public class BackendDeveloper implements Developer {
    @Override
    public void develop() {
        codeJava();
        // codeC();
    }

    private void codeJava() {};
    private void codeC() {};
}

public class FrontendDeveloper implements Developer {

    @Override
    public void develop() {
        codeJS();
        // codeAngular();
    }

    private void codeJS() {};
    private void codeAngular() {};
}

public class Project {

    private List<Developer> developers;

    public Project(List<Developer> developers) {
        this.developers = developers;
    }

    public void build() {
        developers.forEach(developer -> developer.develop());
    }
}
```



SOLID Summary



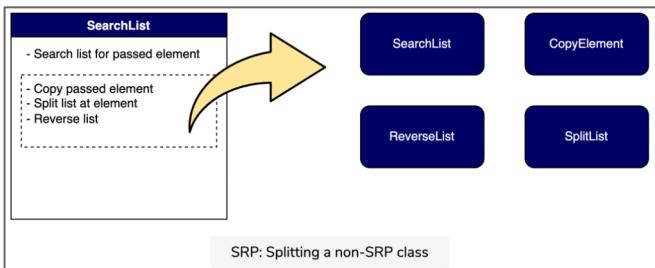
Principles for maintainable object-oriented code



Single Responsibility Principle

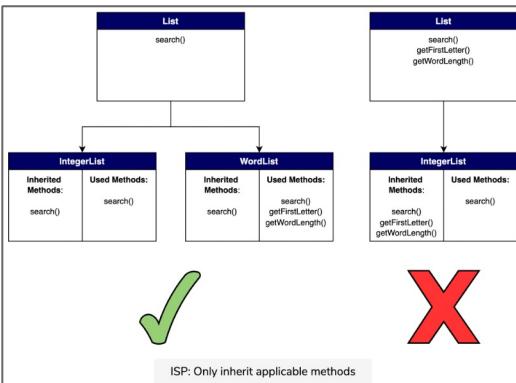
Each class has a single purpose. All its methods should relate to function.
Reasoning: Each responsibility could be a reason to change a class in the future. Fewer responsibilities → fewer opportunities to introduce bugs during changes.

Example: Split formatting & calculating of a report into different classes.



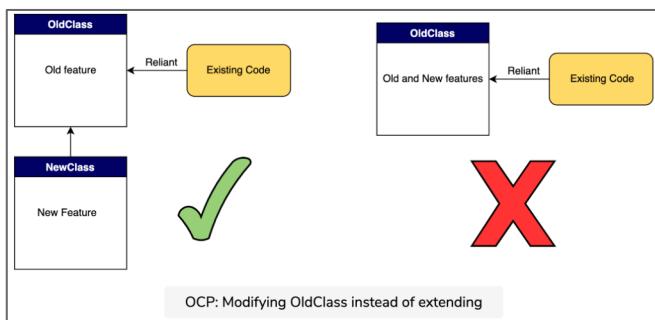
Open / Closed Principle

Classes (or methods) should be open for extension and closed for modification. Once written they should only be touched to fix errors. New functionality should go into new classes that are derived. This is popularly interpreted to advocate inheriting from an abstract base class.
Reasoning: Again you lower the odds of breaking existing code.



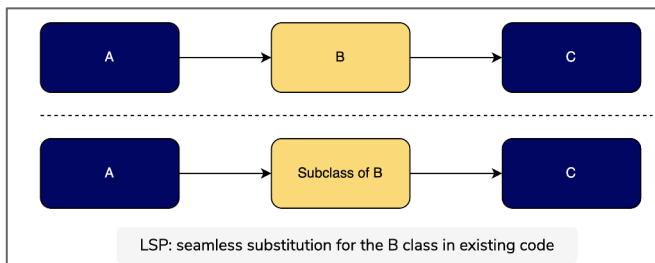
Liskov Substitution Principle

You should be able to replace an object with any of its derived classes. Your code should never have to check which sub-type it's dealing with.
Reasoning: Prevents awkward type checking and weird side-effects.



Interface Segregation Principle

Define subsets of functionality as interfaces.
Reasoning: Small, specific interfaces lead to a more decoupled system than a big general-purpose one.
Example: A PersistenceManager implements DBReader & DBWriter.

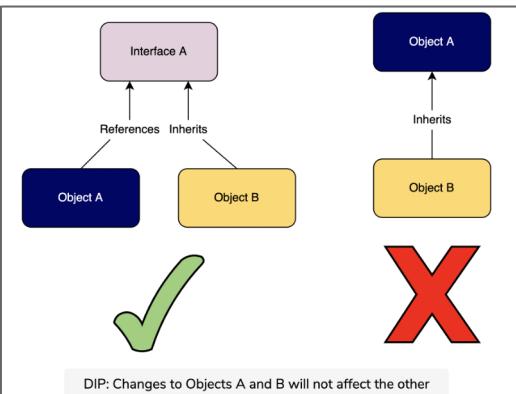


Dependency Inversion Principle

High level modules should not depend on low-level modules. Instead, both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions

Reasoning: High-level modules become more reusable if they are ignorant of low-level module implementation details.

Examples: 1) Dependency Injection. 2) Putting high-level modules in different packages than the low-level modules it uses.



Recap

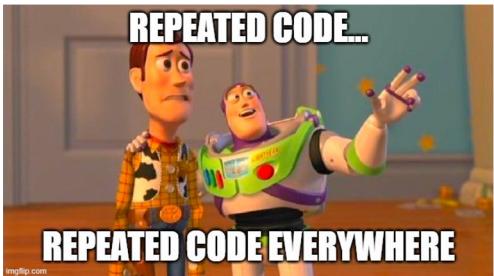


- Object Oriented design principles: **SOLID**
 - S: Single responsibility
 - O: Open/closed principle
 - L: Liskov substitution principle
 - I: Interface segregation principle
 - D: Dependency inversion principle

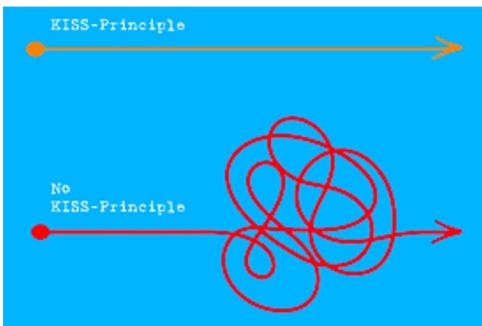


Other Popular Principles

- DRY: “Don’t Repeat Yourself”



- KISS: “Keep It Simple Stupid”



Thank you for your listening!

“Motivation is what gets you started.
Habit is what keeps you going!”

Jim Ryun

