الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

مقرر تصميم نظم البرمجيات | كلية الهندسة المعلوماتية | المحاضرة الثانية

Designing for Simplicity: Principles

SOLID Principles

د. رياض سنبل

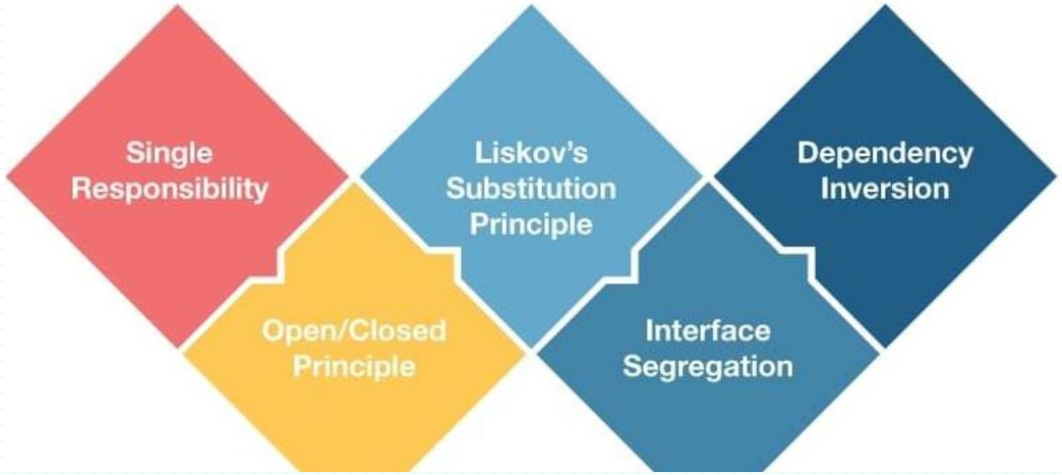
Quick Overview

- Concrete Class vs Abstract Class vs Interface
- Generic Programming (Templates)

SOLID Principles

SOLID principles

- Design principles encourage us to create **more maintainable, understandable, and flexible software**. Consequently, as our applications grow in size, we can **reduce their complexity** and save ourselves a lot of headaches further down the road!
- FIVE Principles
  - **SRP** – Single Responsibility Principle
  - **OCP** – Open/Closed Principle
  - **LSP** – Liskov Substitution Principle
  - **ISP** – Interface Segregation Principle
  - **DIP** – Dependency Inversion Principle
- Why?
  - More understandable code designs
  - Easier to maintain
  - Easier to extend



1st Case

Multiple responsibilities!
Lots of reasons to change!
Difficult to maintain!

What design problem exist?

What limitations stem from these issues?

How can they be addressed or improved?

```
class Order
{
    public function calculateTotalSum(){ }
    public function getItems(){ }
    public function getItemCount(){ }
    public function addItem($item){ }
    public function deleteItem($item){ }

    public function printOrder(){ }
    public function showOrder(){ }

    public function load(){ }
    public function save(){ }
    public function update(){ }
    public function delete(){ }
}
```

Single Responsibility Principle (SRP)

- Do one and only **ONE** thing.. But do it **well** ☺

```
class Order
{
    public function calculateTotalSum(){ }
    public function getItems(){ }
    public function getItemCount(){ }
    public function addItem($item){ }
    public function deleteItem($item){ }
}
```

```
class OrderRepository
{
    public function load($orderID){ }
    public function save($order){ }
    public function update($order){ }
    public function delete($order){ }
}
```

```
class OrderViewer
{
    public function printOrder($order){ }
    public function showOrder($order){ }
}
```

Single Responsibility Principle (SRP)

- The Single Responsibility Principle states that **every object should have a single responsibility**, and that responsibility should be entirely **encapsulated by the class**.

> *"There should never be more than one reason for a class to change."*
> Robert C.Martin

- Classic violations
  - Objects that can print/draw themselves
  - Objects that can save/restore themselves
- Classic solution
  - Separate printer
  - Separate saver (or memento)

2st Case

What if we want to load the data from API from a third party api server?

Difficult to extend

Difficult to reuse

What design problem exist?

What limitations stem from these issues?

How can they be addressed or improved?

```
class OrderRepository
{
    public function load($orderID)
    {
        return DB:table('order')->findorFail($orderID);
    }

    public function save($order){ }
    public function update($order){ }
    public function delete($order){ }
}
```

Open / Closed Principle

Feel Free to **extend** BUT do **not modify**

```
class OrderRepository
{
    private $source;

public function setSource(OrderSource $source)
{
    $this->source = $source;
}

public function load($orderID)
{
    return $this->source->load($orderID);
}

    public function save($order){ }
    public function update($order){ }
}
```

```
interface OrderSource
{
    public function load($orderID);
    public function save($order);
    public function update($order);
    public function delete($order);
}
```

```
class DbOrderSource implements OrderSource
{
    public function load($orderID);
    public function save($order){ }
    public function update($order){ }
    public function delete($order){ }
}
```

```
class ApiOrderSource implements OrderSource
{
    public function load($orderID);
    public function save($order){ }
    public function update($order){ }
    public function delete($order){ }
}
```

Open / Closed Principle

- The Open / Closed Principle states that software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**.
- **Change behavior without changing code?!**
  - Rely on abstractions, not implementations
  - Do not limit the variety of implementations
- **Three approaches to achieve OCP**
  - Parameters
    - Pass delegates / callbacks
  - Inheritance / Template Method pattern
    - Child types override behavior of a base class
  - Composition / Strategy pattern
    - Client code depends on abstraction
    - "Plug in" model

3st Case

What design problem exist?

What limitations stem from these issues?

How can they be addressed or improved?

```php
class Rectangle
{
    public function setWidth($w)
    {
        $this->width = $w;
    }

    public function setHeight($h)
    {
        $this->height = $h;
    }

    public function getArea()
    {
       return $this->height * $this->width;
    }
}
```

Is it a correct Inheritance!

If our work is correct, then we should be able to change Rectangle by Square class

```php
class Square extends Rectangle
{
    public function setWidth($w)
    {
        $this->width = $w;
        $this->height = $w;
    }

    public function setHeight($h)
    {
        $this->height = $h;
        $this->width = $h;
    }
}
```

```php
$rectangle = new Rectangle();
$r->setWidth(7); $r->setHeight(3);
$r->getArea(); // 21
```

Liskov Substitution Principle

If you use base type, you should be able to **use subtypes** and **do not break** anything

```
interface Polygon
{
    public function setHeight($h);
    public function setWidht($w);
    public function getArea();
}

class Rectangle implements Polygon { };

class Square implements Polygon { };
```

Liskov Substitution Principle

- The Liskov Substitution Principle states that in an inheritance, a parent class should be **substitutable for its child class without any problem**.
- In Liskov Subtitution Principle you need to follow the correct hierarchy for your classes **if you do not follow it the unit test** for the superclass would never success for the subclasses.
- **Child classes must not**
  - Remove base class behavior
  - Violate base class invariants
- If an override method does nothing or throws and exception, you're probably violating LSP.

4st Case

- Difficult to reuse
- Interface is to big too implement
- Potential violation of single responsibility

What design problem exist?

What limitations stem from these issues?

How can they be addressed or improved?

```
interface Product
{
    public function applyDiscount($discount);
    public function applyPromocode($promocode);

    public function setColor($color);
    public function setSize($size);

    public function setCondition($condition);
    public function setPrice($price);
}
```

Interface segregation principle

Several specialized interfaces are **better** than **One All-Purpose interface**.

```
interface Product
{
    public function setCondition($condition);
    public function setPrice($price);
}

interface Clothes
{
    public function setColor($color);
    public function setSize($size);
    public function setMaterial($material);
}

interface Discountable
{
    public function applyDiscount($discount);
    public function applyPromocode($promocode);
}
```

```
class Book implemets Product, Discountable
{
    public function setCondition($condition){ }
    public function setPrice($price){ }
    public function applyDiscount($discount){ }
    public function applyPromocode($promocode){ }
}

class MenClothes implemets Product, Clothes
{
    public function setCondition($condition){ }
    public function setPrice($price){ }
    public function setColor($color){ }
    public function setSize($size){ }
    public function setMaterial($material){ }
}
```

Interface segregation principle

- The Interface Segregation Principle states that Clients should not be forced to depend on methods they do not use.
- Divide "fat" interfaces into smaller ones.

5st Case

Low Level Operations

```
class worker
{
  public function work(){ }

}
```

High Level Operations

```
class Manager{

 private $worker;

 public function setWorker($worker)
 {
    $this->worker = $worker;
 }

 public function manager()
 {
    $this->worker->work();
 }

}
```

- Less Flexible (WHY?)
- Difficult to write unit tests

What design problem exist?

What limitations stem from these issues?

How can they be addressed or improved?

Dependency Inversion Principle

- Depend on abstraction not implementation.
- High-level modules should not depend on low-level modules. Both should depend on abstractions.

```
interface Employee{

public function work(){}
}

class Worker implements Employee
{
   public function work(){}
}

class SpecializedWorker implements Employee
{
   public function work(){}
}
```