



الجامعة السورية الخاصة  
SYRIAN PRIVATE UNIVERSITY

المحاضرة الثانية

كلية الهندسة المعلوماتية

مقرر تصميم نظم البرمجيات

# Designing for Simplicity: Principles

## *SOLID Principles*

د. رياض سنبل

# Quick Overview

---

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

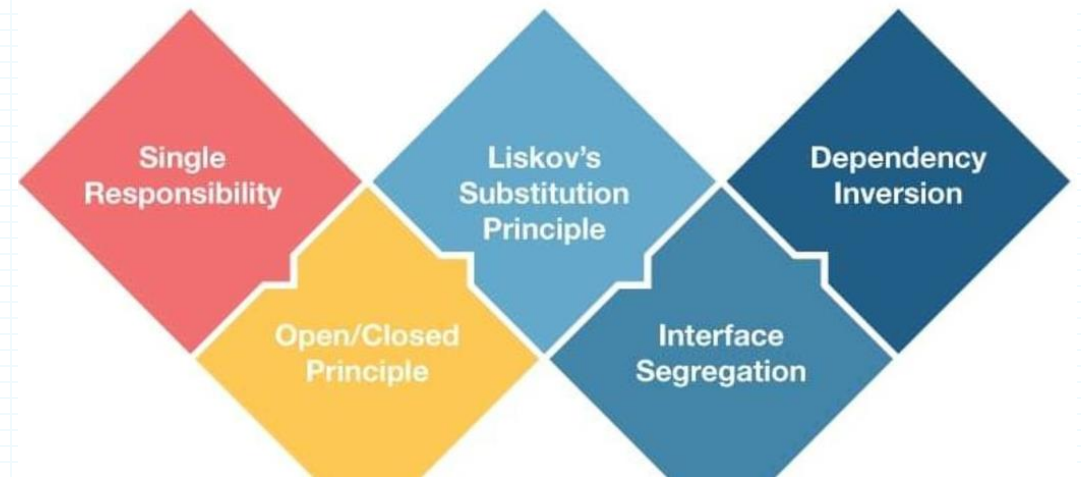
# SOLID Principles

---

# 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



# 1<sup>st</sup> Case



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(){ }
}
```

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

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

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

Robert C. Martin

## 2<sup>st</sup> Case



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){ }
}
```

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

# 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?

```
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

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

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

```
$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 Substitution 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 an exception, you're probably violating LSP.

# 4<sup>st</sup> Case



**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);
}
```

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

# 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 implements Product, Discountable
{
    public function setCondition($condition){ }
    public function setPrice($price){ }
    public function applyDiscount($discount){ }
    public function applyPromocode($promocode){ }
}

class MenClothes implements 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.

# 5<sup>st</sup> Case



Low Level  
Operations

High Level  
Operations

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

```
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(){}  
}
```

# Examples - SOLID

---

# Example 1

---

```
public class Invoice {  
    private String customer;  
    private String state;  
    private int total;  
  
    public Invoice(String customer, String state, int total) {  
        this.customer = customer;  
        this.state = state;  
        this.total = total;  
    }  
  
    //Getters & Setters  
  
    public String getDetails() {  
        return this.getCustomer() + ", " + this.getTotal();  
    }  
  
    public void emailInvoice() {  
        System.out.println("Sending email...");  
        System.out.println(this.details());  
    }  
  
    public double determineTax() {  
        switch(this.state) {  
            case "VIC":  
                return 0.3;  
            case "NSW":  
                return 0.5;  
            default:  
                return;  
        }  
    }  
}
```

# Example 1

---

This violates the SRP principle.

## Refactoring

We'll be creating separate classes:

- **Mailer** — *handles the emailing*
- **Sales Tax** — *handles the sales tax calculation*
- **Invoice** — *handles the invoice info*

```
public class Invoice {  
  
    private String customer;  
    private String state;  
    private int total;  
  
    public Invoice(String customer, String state, int total) {  
        this.customer = customer;  
        this.state = state;  
        this.total = total;  
    }  
  
    //Getters & Setters  
  
    public String getDetails() {  
        return this.getCustomer() + ", " + this.getTotal();  
    }  
  
    public void emailInvoice() {  
        System.out.println("Sending email...");  
        System.out.println(this.details());  
    }  
  
    public double determineTax() {  
        switch(this.state) {  
            case "VIC":  
                return 0.3;  
            case "NSW":  
                return 0.5;  
            default:  
                return;  
        }  
    }  
}
```

# Example 2

```
abstract class Employee {  
    private Long id;  
    private String name;  
    private double salary;  
  
    //Constructor  
  
    //Getters & Setters  
  
    abstract void getBonusAmount();  
  
}
```

```
public static void main(String[] args) {  
    //Adding employees  
    Employee a = new PermanentEmployee(1, "John Doe", 50000);  
    Employee b = new TemporaryEmployee(1, "Jane Citizen", 60000);  
    Employee c = new ContractEmployee(1, "Joe Bloggs", 75000);  
  
    Employee[] employees = {a, b, c};  
  
    for(Employee e : employees) {  
        e.getBonusAmount(); //Throws exception for c object  
    }  
}
```

```
public class PermanentEmployee extends Employee {  
    //Constructor  
  
    //Getters & Setters  
  
    public void getBonusAmount() {  
        System.out.println(this.getSalary() * 0.1);  
    }  
}  
  
public class TemporaryEmployee extends Employee {  
    //Constructor  
  
    //Getters & Setters  
  
    public void getBonusAmount() {  
        System.out.println(this.getSalary() * 0.1);  
    }  
}  
  
public class ContractEmployee extends Employee {  
    //Constructor  
  
    //Getters & Setters  
  
    public void getBonusAmount() throws Exception {  
        throw new InvalidEmployeeException();  
    }  
}
```

# Example 2

---

- Refactoring our code
  - Convert Employee to a concrete class
  - Create an IBonus interface with the getBonusAmount() method
  - Have only the classes that are eligible to receive a bonus to implement IBonus

```
class Employee {  
  
    private Long id;  
    private String name;  
    private double salary;  
  
    //Constructor  
  
    //Getters & Setters  
  
}
```

```
public interface IBonus {  
    public void getBonusAmount();  
}
```

# Example 2

- Refactoring our code
  - Convert Employee to a concrete class
  - Create an IBonus interface with the getBonusAmount() method
  - Have only the classes that are eligible to receive a bonus to implement IBonus

```
public class PermanentEmployee extends Employee implements IBonus {  
    //Constructor  
    //Getters & Setters  
  
    @Override  
    public void getBonusAmount() {  
        System.out.println("Bonus for perm employee: " +  
this.getSalary() * 0.1);  
    }  
}  
  
public class TemporaryEmployee extends Employee implements IBonus {  
    //Constructor  
    //Getters & Setters  
  
    @Override  
    public void getBonusAmount() {  
        System.out.println("Bonus for temp employee: " +  
this.getSalary() * 0.15);  
    }  
}  
  
public class ContractEmployee extends Employee {  
    //Constructor  
    //Getters & Setters  
    //Other functionality  
  
}
```

# Example 2

---

```
public static void main(String[] args) {  
    //Create our objects  
    TemporaryEmployee temp = new TemporaryEmployee("John", 60000);  
    PermanentEmployee perm = new PermanentEmployee("Bob", 50000);  
    ContractEmployee con = new ContractEmployee("Jack", 75000);  
  
    //Create a list of IBonus interface type  
    List<IBonus> employees = new ArrayList<IBonus>();  
  
    employees.add(temp);  
    employees.add(perm);  
    /* employees.add(con); compilation error: ContractEmployee doesn't  
    implement IBonus interface */  
  
    for(IBonus e : employees) {  
        e.getBonusAmount();  
    }  
  
    /*  
    For-loop returns:  
    Bonus for temp employee: $9000  
    Bonus for perm employee: 5000  
    */  
}
```