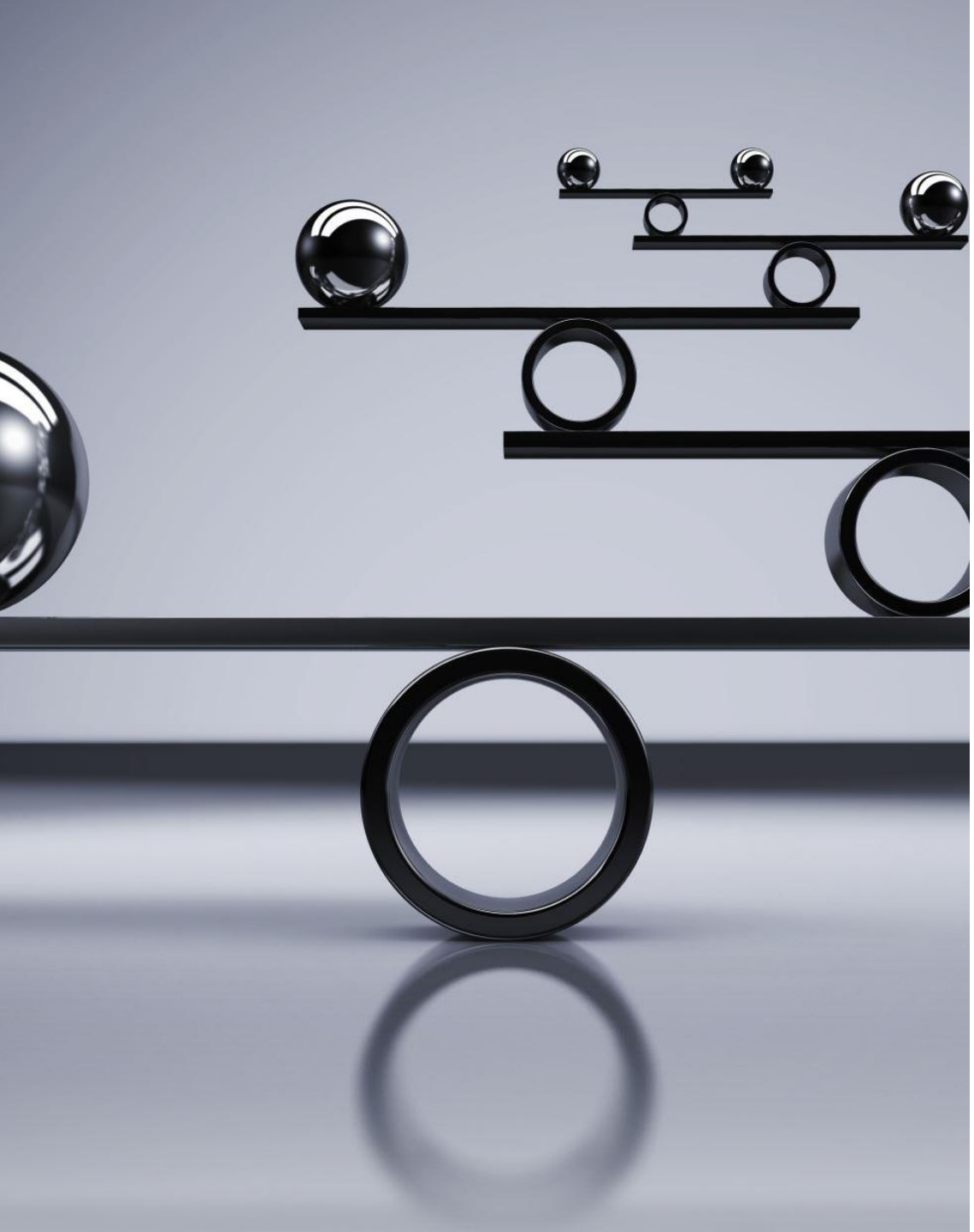
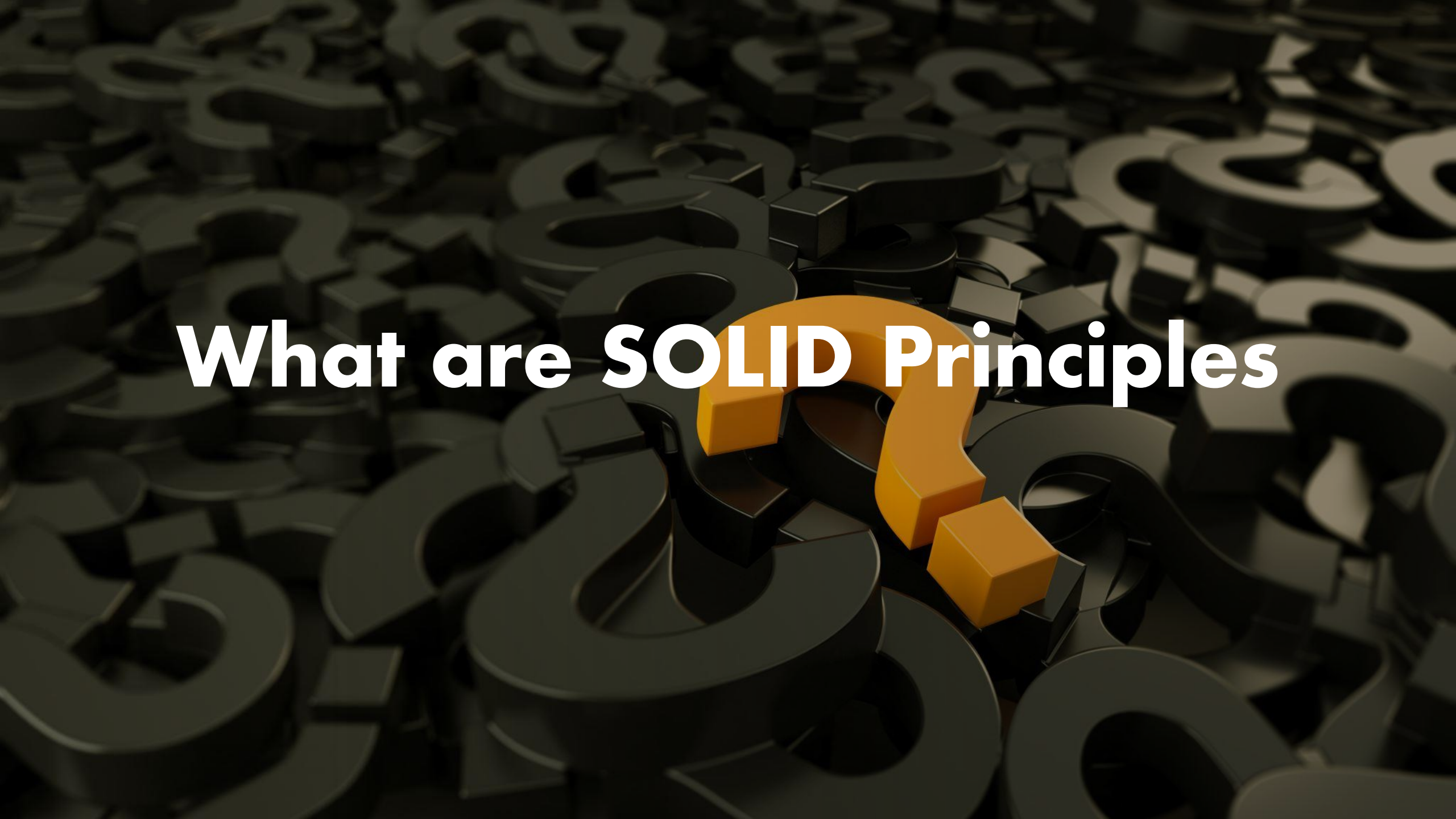
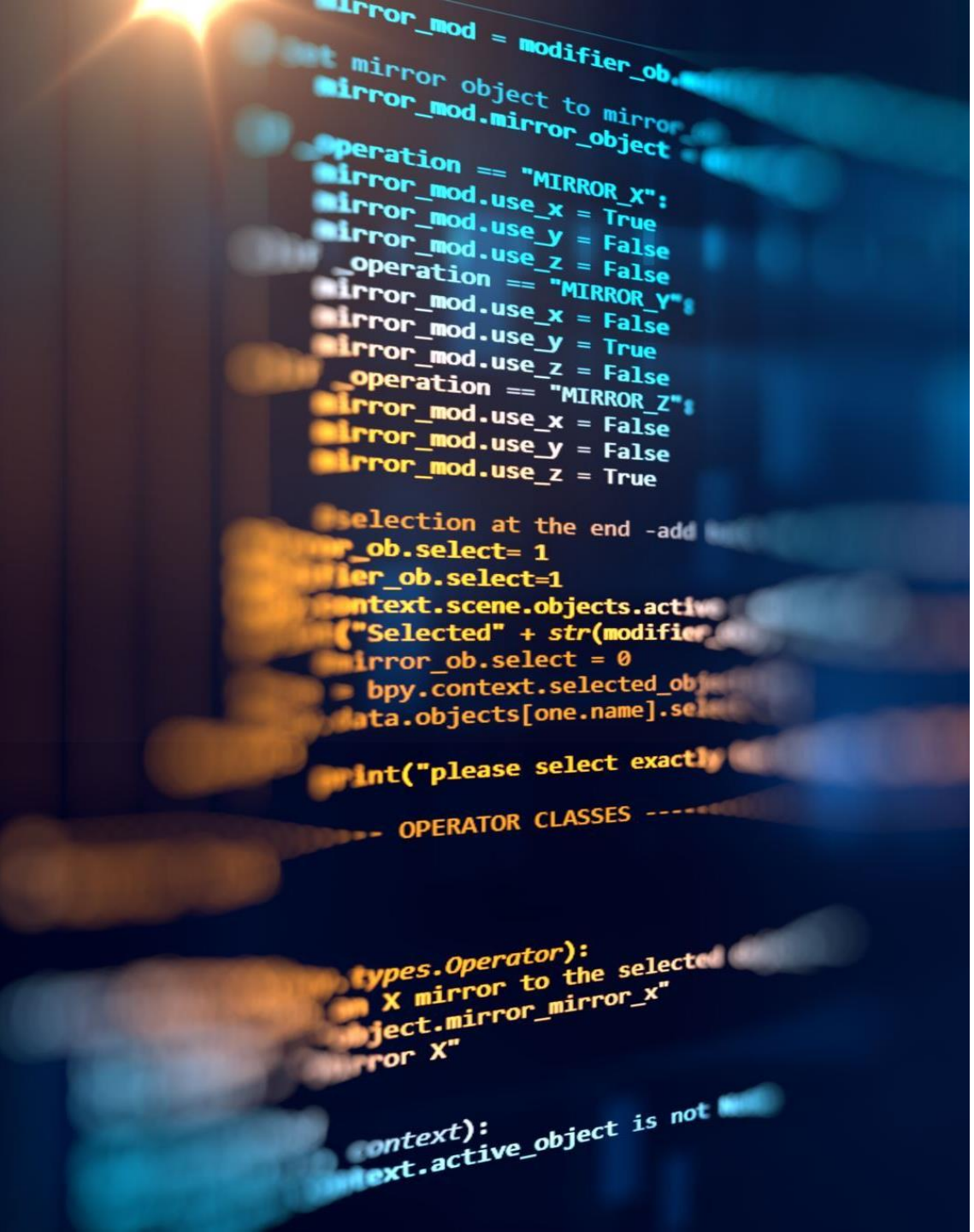


SOLID Principles

With PHP

Part 1


What are SOLID Principles



SOLID is an acronym that stands for five key design principles:

Single responsibility principle
Open-closed principle
Liskov substitution principle
Interface segregation principle
Dependency inversion principle.

All five are commonly used by software engineers and provide some important benefits for developers.

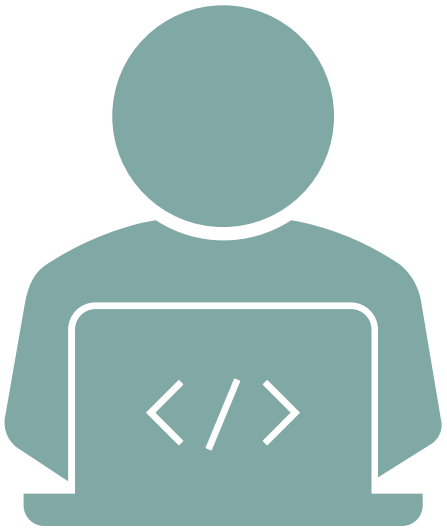
Single Responsibility Principle

- What is SRP and why it's important
- Code that violates SRP
- Code to adhere to SRP

Robert Martin summarizes this principle well by mandating that, “a class should have one, and only one, reason to change.” Following this principle means that each class only does one thing, and every class or module only has responsibility for one part of the software’s functionality. More simply, each class should solve only one problem.

Single responsibility principle is a relatively basic principle that most developers are already utilizing to build code. It can be applied to classes, software components, and microservices.

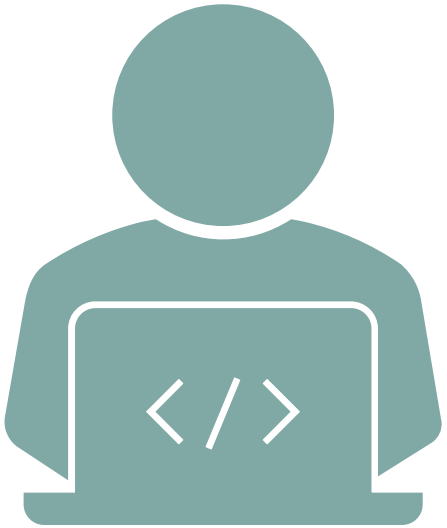
Code Example



```
class Order
{
    public function calculateTotalPrice($items)
    {
        $total = 0;
        foreach ($items as $item) {
            $total += $item->price;
        }
        return $total;
    }

    public function sendConfirmationEmail($user)
    {
        $message = "Thank you for your order!";
        mail($user->email, "Order Confirmation", $message);
    }
}
```

This code violates SRP because the **Order** class is responsible for both calculating the total price of an order and sending a confirmation email. To adhere to SRP, we should separate these responsibilities into separate classes.

Code Example



```php
class OrderCalculator
{
    public function calculateTotalPrice($items)
    {
        $total = 0;
        foreach ($items as $item) {
            $total += $item->price;
        }
        return $total;
    }
}

class EmailSender
{
    public function sendConfirmationEmail($user)
    {
        $message = "Thank you for your order!";
        mail($user->email, "Order Confirmation", $message);
    }
}
```

Open-Closed Principle (OCP)

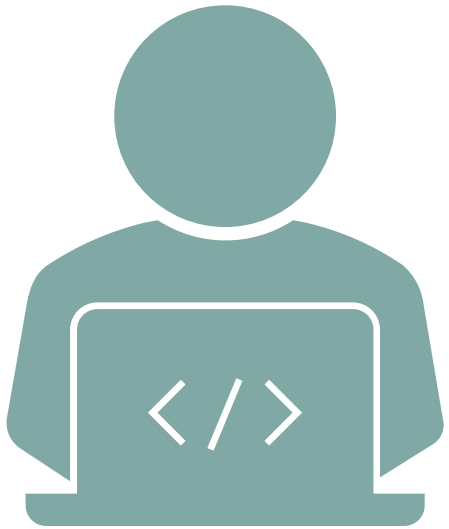
- What is OCP and why it's important
- Code that violates OCP
- Code to adhere to OCP

The idea of open-closed principle is that existing, well-tested classes will need to be modified when something needs to be added. Yet, changing classes can lead to problems or bugs. Instead of changing the class, you simply want to extend it. With that goal in mind, Martin summarizes this principle, "You should be able to extend a class's behavior without modifying it."

Following this principle is essential for writing code that is easy to maintain and revise. Your class complies with this principle if it is:

- Open for extension, meaning that the class's behavior can be extended; and
- Closed for modification, meaning that the source code is set and cannot be changed.

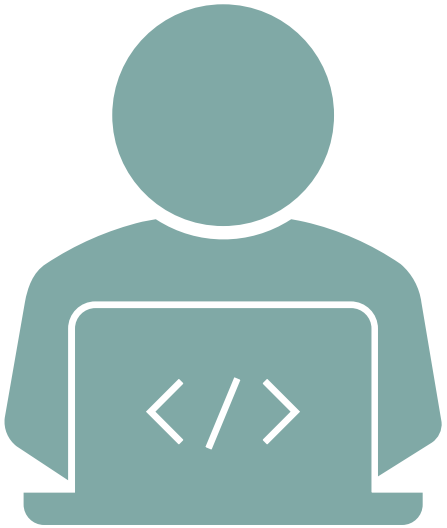
Code Example



```
class Order
{
    public function calculateTotalPrice($items, $discount)
    {
        $total = 0;
        foreach ($items as $item) {
            $total += $item->price;
        }
        $total *= (1 - $discount);
        return $total;
    }
}
```

This code violates OCP because it's not open for extension but closed for modification. If we want to add a new type of discount, we will have to modify the **Order** class. To adhere to OCP, we should use abstraction and inheritance.

Code Example



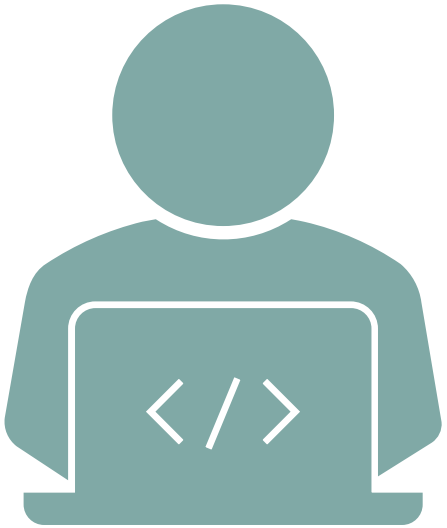
```php
abstract class Discount
{
    public abstract function apply($total);
}

class PercentageDiscount extends Discount
{
    private $percentage;

    public function __construct($percentage)
    {
        $this->percentage = $percentage;
    }

    public function apply($total)
    {
        return $total * (1 - $this->percentage);
    }
}

class Order
{
    public function calculateTotalPrice($items, $discount)
    {
        $total = 0;
        foreach ($items as $item) {
            $total += $item->price;
        }
        $total = $discount->apply($total);
        return $total;
    }
}
```

Liskov Substitution Principle (LSP)

Of the five SOLID principles, the Liskov Substitution Principle is perhaps the most difficult one to understand. Broadly, this principle simply requires that every derived class should be substitutable for its parent class. The principle is named for Barbara Liskov, who introduced this concept of behavioral subtyping in 1987. Liskov herself explains the principle by saying:

What is wanted here is something like the following substitution property: if for each object O1 of type S there is an object O2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O1 is substituted for O2 then S is a subtype of T

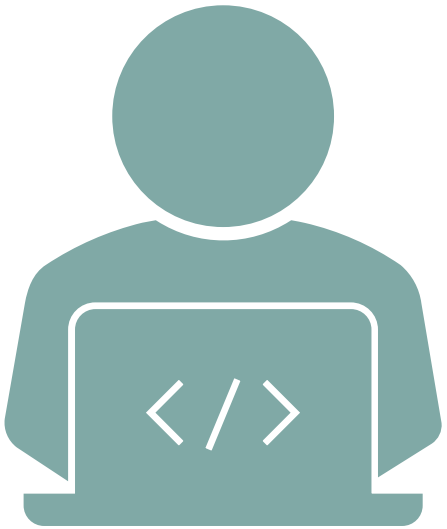
Code Example



```
class Rectangle
{
    protected $width;
    protected $height;

    public function setWidth($width)
    {
        $this->width = $width;
    }

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

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

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

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

This code violates LSP because a **Square** is not substitutable for a **Rectangle**. If we set the width and height of a **Square** to different values, the **area()** method will not return the expected result. To adhere to LSP, we should use composition instead of inheritance.

Code Example



```php
<?php

class Shape {
    protected $width;
    protected $height;

    public function setWidth($width) {
        $this->width = $width;
    }

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

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

class Rectangle extends Shape {
    // No need to add anything here, as this class already satisfies the LSP.
}

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

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

function printArea(Shape $shape) {
    $shape->setWidth(5);
    $shape->setHeight(10);
    echo "The area of the shape is " . $shape->area() . "</br>";
}

$rectangle = new Rectangle();
printArea($rectangle); // The area of the shape is 50

$square = new Square();
printArea($square); // The area of the shape is 100
```

Confused?

Let's Try Again

Bad Example of LSP:

```
public class Bird{
 public void fly(){}
}
public class Duck extends Bird{}
```

The duck can fly because it is a bird, But what about this:

```
public class Ostrich extends Bird{}
```

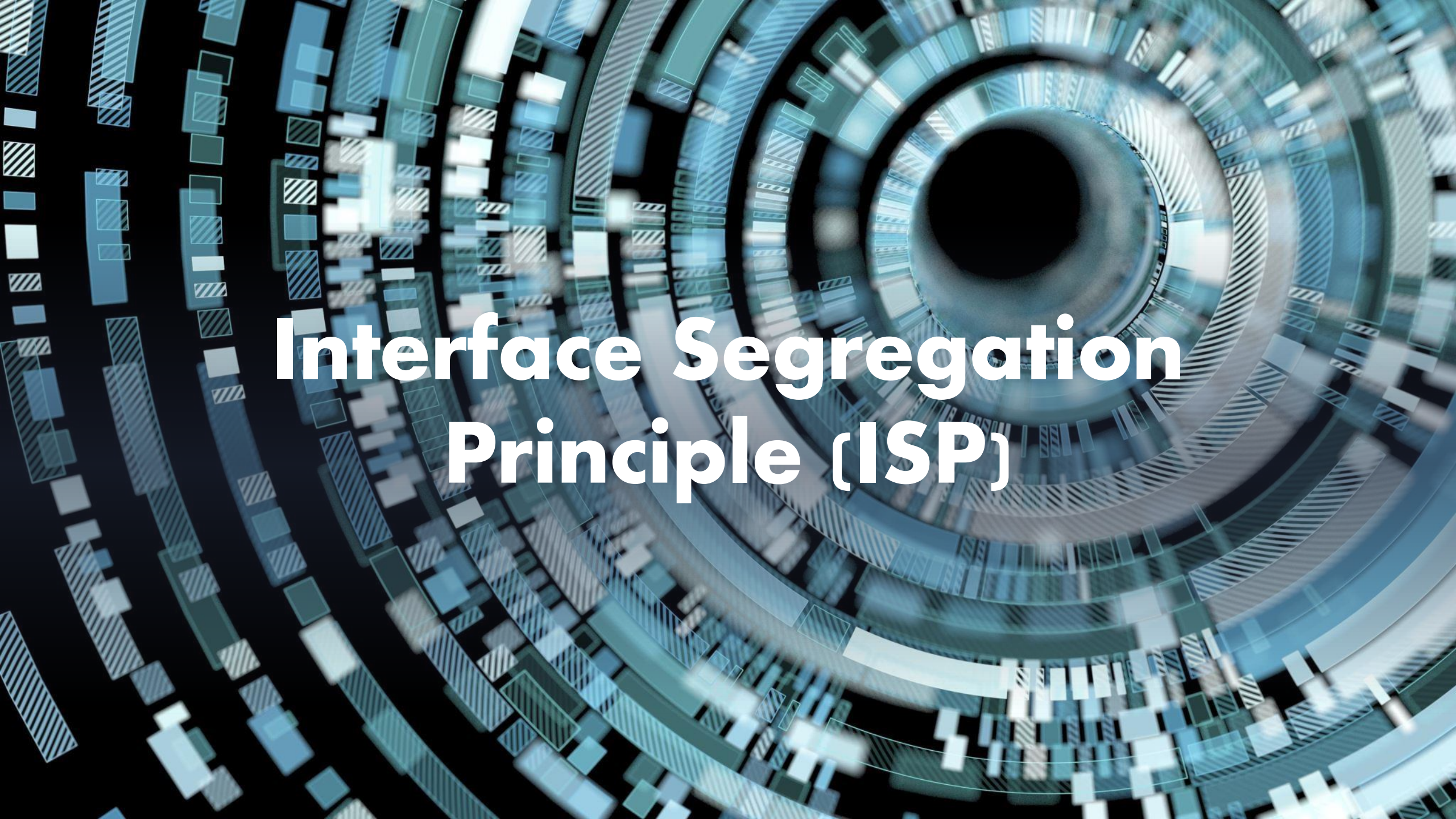
Ostrich is a bird, But it can't fly, Ostrich class is a subtype of class Bird, But it **can't use the fly method**, that means that we are breaking the **LSP** principle.

Let's Fix It

Good Example of LSP:

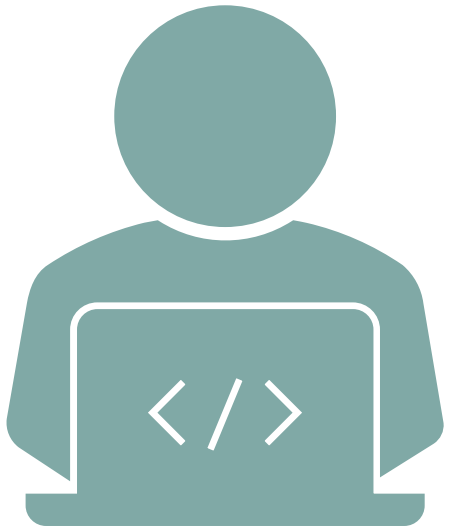
```
public class Bird{
}
public class FlyingBirds extends Bird{
    public void fly(){}
}
public class Duck extends FlyingBirds{}
public class Ostrich extends Bird{}
```

> *ability to replace any instance of a parent class with an instance of one of its child classes* ***without negative side effect.***


Interface Segregation
Principle (ISP)

The Interface Segregation Principle (ISP) states that clients should not be forced to depend on interfaces they do not use. In other words, we should create small, focused interfaces that only contain the methods that a client needs. Here's an example of PHP code that violates the ISP:

Code Example

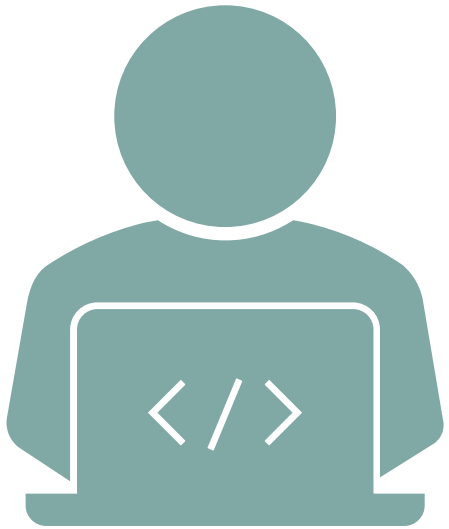


```
interface Animal {
    public function eat();
    public function fly();
    public function swim();
}

class Bird implements Animal {
    public function eat() {
        // ...
    }

    public function fly() {
        // ...
    }

    public function swim() {
        throw new Exception('Birds cannot swim');
    }
}

class Fish implements Animal {
    public function eat() {
        // ...
    }

    public function fly() {
        throw new Exception('Fish cannot fly');
    }

    public function swim() {
        // ...
    }
}
```

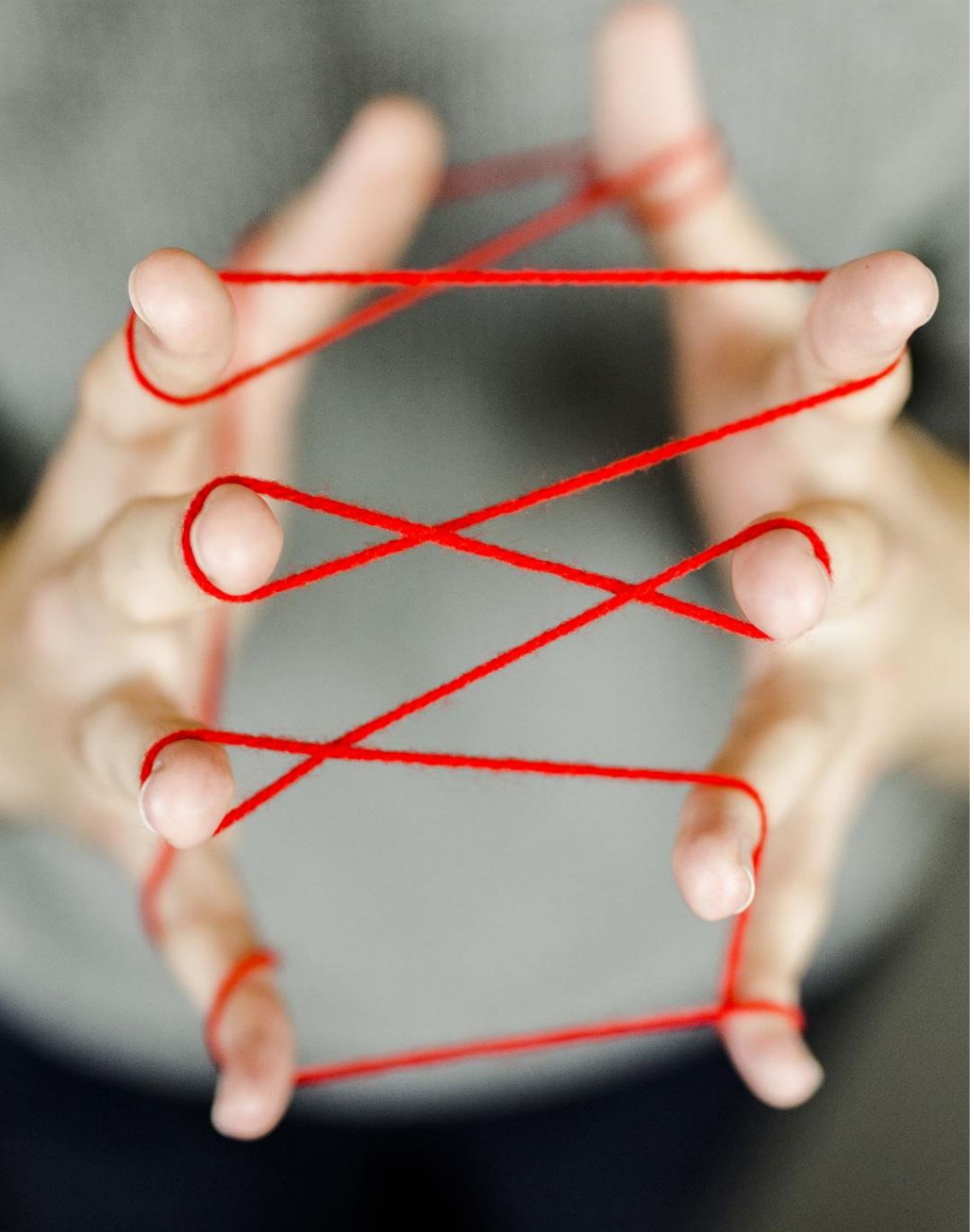
In this example, the **Animal** interface contains three methods: **eat()**, **fly()**, and **swim()**. The **Bird** class implements all three methods, but it throws an exception for the **swim()** method because birds cannot swim. The **Fish** class also implements all three methods, but it throws an exception for the **fly()** method because fish cannot fly. This violates the ISP because clients (i.e., classes that use the **Animal** interface) may be forced to implement methods they do not need. For example, a client that only needs to work with birds may be forced to implement the **swim()** method, even though it is not relevant.

To fix this violation, we can split the **Animal** interface into smaller, more focused interfaces:

Code Example



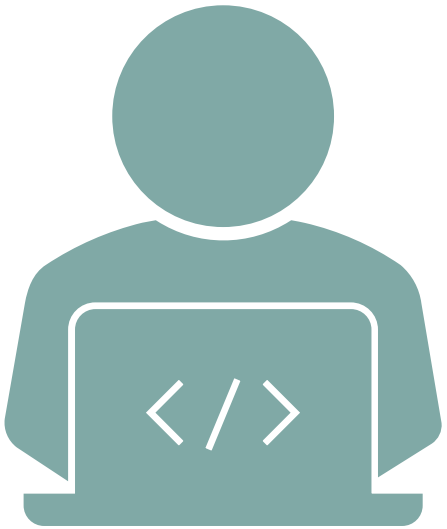
```
interface Bird {
    public function eat();
    public function fly();
}

interface Fish {
    public function eat();
    public function swim();
}

class Sparrow implements Bird {
    public function eat() {
        // ...
    }

    public function fly() {
        // ...
    }
}

class Salmon implements Fish {
    public function eat() {
        // ...
    }

    public function swim() {
        // ...
    }
}
```



Dependency Inversion Principle

High level modules should not depend upon low level modules. Both should depend on abstractions." Further, "abstractions should not depend on details. Details should depend upon abstractions.

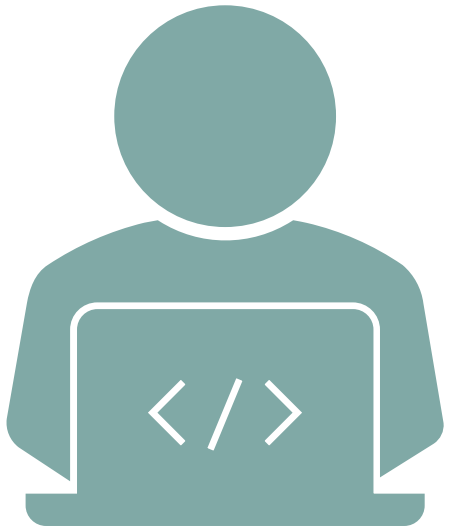
Code Example



```
class UserService {
    private $userRepository;

    public function __construct() {
        $this->userRepository = new UserRepository();
    }

    public function getUserById($userId) {
        return $this->userRepository->getUserById($userId);
    }
}

class UserRepository {
    public function getUserById($userId) {
        // Query database to retrieve user data
        $user = // ...
        return $user;
    }
}
```

In this example, the **UserService** class depends directly on the **UserRepository** class, which is a low-level module. This violates the DIP because high-level modules should not depend on low-level modules.
To fix this violation, we can introduce an abstraction between the **UserService** and **UserRepository** classes:

Code Example



```
interface UserRepositoryInterface {
    public function getUserById($userId);
}

class UserRepository implements UserRepositoryInterface {
    public function getUserById($userId) {
        // Query database to retrieve user data
        $user = // ...
        return $user;
    }
}

class UserService {
    private $userRepository;

    public function __construct(UserRepositoryInterface
$userRepository) {
        $this->userRepository = $userRepository;
    }

    public function getUserById($userId) {
        return $this->userRepository->getUserById($userId);
    }
}
```

Questions

Conclusion