SOLID

Workshop AFUP Montpellier

Younes BOUKOBAA

13 décembre 2025

Les Principes de Conception SOLID

Introduit par Robert C. Martin ("Uncle Bob")

5 lignes directrices pour la Programmation Orientée Objet (POO)

Objectifs du code produit :

✅ Maintenable ✅ Extensible ✅ Robuste

Ce ne sont **pas des règles strictes**, mais des **bonnes pratiques** pour créer un code **robuste** et **maintenable**.

S R P

Single Responsibility Principle

“Une classe ne devrait avoir qu'une seule raison de changer”

```php
<?php

class OrderManager {
    public function createOrder() { /* ... */ }
    public function sendOrderEmail() { /* ... */ }
    public function generateOrderPDF() { /* ... */ }
    public function saveOrderLog() { /* ... */ }
}
```

```php
<?php

class OrderManager {
    public function createOrder() { /* ... */ }
    public function sendOrderEmail() { /* ... */ }
    public function generateOrderPDF() { /* ... */ }
    public function saveOrderLog() { /* ... */ }
}
```

- *4 responsabilités → 4 raisons de changer*
- *Couplage fort*
- *Difficile à tester*

```php
<?php

class OrderCreator {
    public function create() { /* ... */ }
}

class EmailNotifier {
    public function send() { /* ... */ }
}

class PdfInvoiceGenerator {
    public function generate() { /* ... */ }
}

class OrderLogger {
    public function save() { /* ... */ }
}
```

SRP

Rend le code plus facile

```php
<?php

class OrderCreator {
    public function create() { /* ... */ }
}

class EmailNotifier {
    public function send() { /* ... */ }
}

class PdfInvoiceGenerator {
    public function generate() { /* ... */ }
}

class OrderLogger {
    public function save() { /* ... */ }
}
```

- *à comprendre*
- *à tester*
- *à maintenir en isolant les responsabilités.*

Code qui marche ≠ code maintenable

O C P

Open/Closed Principle

“Les entités logicielles (classes, modules, fonctions, etc.)
devraient être ouvertes à l'extension,
mais fermées à la modification”

```php
<?php

class PaymentProcessor {
    public function process(string $type, float $amount) {
        if ($type === 'stripe') {
            // paiement via Stripe
        } else if ($type === 'paypal') {
            // paiement via Paypal
        }
    }
}
```

```php
<?php

class PaymentProcessor {
    public function process(string $type, float $amount) {
        if ($type === 'stripe') {
            // paiement via Stripe
        } else if ($type === 'paypal') {
            // paiement via Paypal
        }
    }
}
```

- *Fermé à l'extension*
- *Risque de régression*
- *Complexité croissante*

```php
<?php

interface PaymentMethod {
    public function pay(float $amount);
}

class StripePayment implements PaymentMethod {
    public function pay(float $amount) {
        // logique stripe
    }
}

class PaypalPayment implements PaymentMethod {
    public function pay(float $amount) {
        // logique paypal
    }
}

class PaymentProcessor {
    public function process(PaymentMethod $payment, float $amount) {
        $payment->pay($amount);
    }
}
```

OCP

Permet d'ajouter de nouvelles fonctionnalités sans réécrire ou risquer de casser le code existant

```php
<?php

interface PaymentMethod {
    public function pay(float $amount);
}

class StripePayment implements PaymentMethod {
    public function pay(float $amount) {
        // logique stripe
    }
}

class PaypalPayment implements PaymentMethod {
    public function pay(float $amount) {
        // logique paypal
    }
}

class PaymentProcessor {
    public function process(PaymentMethod $payment, float
        $payment->pay($amount);
    }
}
```

- ***Ouvert à l'extension***
- ***Fermé à la modification***
- ***Accessoirement respect du SRP***

L S P

Liskov Substitution Principle

“Un objet d’une classe fille doit pouvoir remplacer un objet de sa classe parente sans casser le comportement attendu.”

```php
class Bird {
    public function fly(): string {
        return "Je vole dans le ciel.";
    }

    public function eat(): string {
        return "Je mange.";
    }
}

class Penguin extends Bird {
    public function fly(): string {
        throw new \Exception("Je ne peux pas voler !");
    }
}

class Eagle extends Bird {
    public function fly(): string {
        return "Je vole très haut.";
    }
}
```

- ***Respecte pas le contrat de la méthode "fly"***
- ***Change le comportement***
- ***Crée un effet de bord***

```php
<?php

class Bird {
    public function fly(): string {
        return "Je prends mon envol !";
    }

    public function eat(): string {
        return "Je mange des graines.";
    }
}

class Penguin extends Bird {
    public function fly(): string {
        throw new \Exception("Je ne peux pas voler !");
    }
}

class Eagle extends Bird {
    public function fly(): string {
        return "Je vole haut dans le ciel !";
    }
}
```

```php
<?php

interface Flyable {
    public function fly(): string;
}

class Bird {
    public function eat(): string {
        return "Je mange.";
    }
}

class Penguin extends Bird {
    public function eat(): string {
        return "Je mange du poisson";
    }
}

class Eagle extends Bird implements Flyable {
    public function fly(): string {
        return "Je vole très haut.";
    }
}
```

LSP

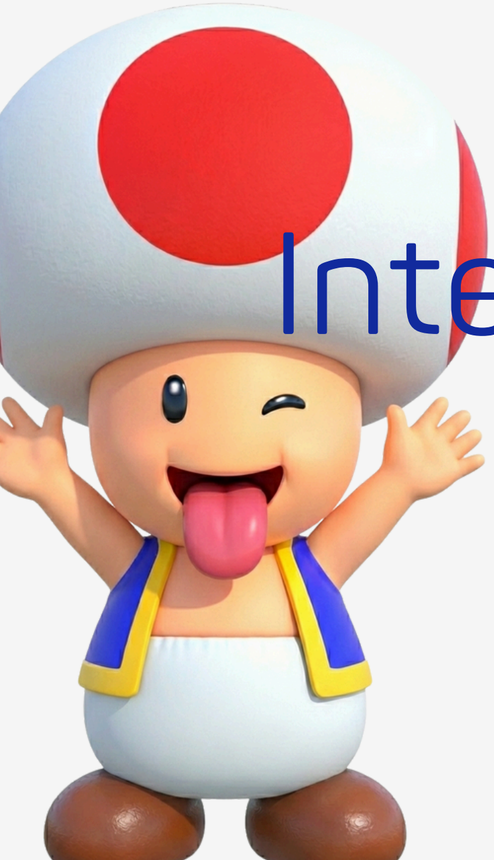
Garantit que l'héritage est utilisé de manière fiable et prévisible, évitant des bugs subtils et renforçant la stabilité du code

```php
<?php

interface Flyable {
    public function fly(): string;
}

class Bird {
    public function eat(): string {
        return "Je mange.";
    }
}

class Penguin extends Bird {
    public function eat(): string {
        return "Je mange du poisson";
    }
}

class Eagle extends Bird implements Flyable {
    public function fly(): string {
        return "Je vole très haut.";
    }
}
```

- ***Prévisibilité***
- ***Pas de substitution incorrecte***
- ***L'héritage est utilisé à bon escient***

I S P

Interface Segregation Principle



“Ne jamais forcer une classe à implémenter des méthodes dont elle n’a pas besoin”

```php
<?php

interface BirdActions {
    public function fly(): string;
    public function swim(): string;
    public function walk(): string;
}

class Eagle implements BirdActions {
    public function fly(): string {
        return "Je vole haut.";
    }

    public function swim(): string {
        throw new \Exception("Je ne peux pas nager !");
    }

    public function walk(): string {
        return "Je marche.";
    }
}

class Penguin implements BirdActions {
    public function fly(): string {
        throw new \Exception("Je ne peux pas voler !");
    }

    public function swim(): string {
        return "Je nage.";
    }

    public function walk(): string {
        return "Je marche.";
    }
}
```

- ***Implémentation forcée***
- ***Dépendances inutiles***
- ***Couplage fort***

```php
<?php

interface BirdActions {
    public function fly(): string;
    public function swim(): string;
    public function walk(): string;
}

class Eagle implements BirdActions {
    public function fly(): string {
        return "Je vole haut.";
    }

    public function swim(): string {
        throw new \Exception("Je ne peux pas nager !");
    }

    public function walk(): string {
        return "Je marche.";
    }
}

class Penguin implements BirdActions {
    public function fly(): string {
        throw new \Exception("Je ne peux pas voler !");
    }

    public function swim(): string {
        return "Je nage.";
    }

    public function walk(): string {
        return "Je marche.";
    }
}
```

```php
<?php

interface CanFly {
    public function fly(): string;
}

interface CanSwim {
    public function swim(): string;
}

interface CanWalk {
    public function walk(): string;
}

class Eagle implements CanFly, CanWalk {
    public function fly(): string {
        return "Je vole haut.";
    }

    public function walk(): string {
        return "Je marche.";
    }
}

class Penguin implements CanSwim, CanWalk {
    public function swim(): string {
        return "Je nage.";
    }

    public function walk(): string {
        return "Je marche.";
    }
}
```

ISP

Préférer des interfaces petites et spécifiques à de grosses interfaces générales.

- ***Haute cohésion***
- ***Découplage***
- ***Flexibilité***

```php
<?php

interface CanFly {
    public function fly(): string;
}

interface CanSwim {
    public function swim(): string;
}

interface CanWalk {
    public function walk(): string;
}

class Eagle implements CanFly, CanWalk {
    public function fly(): string {
        return "Je vole haut.";
    }

    public function walk(): string {
        return "Je marche.";
    }
}

class Penguin implements CanSwim, CanWalk {
    public function swim(): string {
        return "Je nage.";
    }

    public function walk(): string {
        return "Je marche.";
    }
}
```

D I P

Dependency Inversion Principle

“Les modules haut niveau ne doivent pas dépendre des modules bas niveau.
Les deux doivent dépendre d'abstractions “

```php
<?php

class EmailService {
    public function sendEmail(string $message): void {
        echo "Email envoyé : $message";
    }
}

class BirdNotifier {
    private EmailService $emailService;

    public function __construct() {
        $this->emailService = new EmailService();
    }

    public function notify(string $msg): void {
        $this->emailService->sendEmail($msg);
    }
}
```

- ***Couplage fort***
- ***Non réutilisable***
- ***Difficile à tester***

```php
<?php

class EmailService {
    public function sendEmail(string $message): void {
        echo "Email envoyé : $message";
    }
}

class BirdNotifier {
    private EmailService $emailService;

    public function __construct() {
        $this->emailService = new EmailService();
    }

    public function notify(string $msg): void {
        $this->emailService->sendEmail($msg);
    }
}
```

```php
<?php

interface NotifierInterface {
    public function send(string $message): void;
}

class EmailNotifier implements NotifierInterface {
    public function send(string $message): void {
        echo "Email envoyé : $message";
    }
}

class SmsNotifier implements NotifierInterface {
    public function send(string $message): void {
        echo "SMS envoyé : $message";
    }
}

class BirdNotifier {
    private NotifierInterface $notifier;

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

    public function notify(string $msg): void {
        $this->notifier->send($msg);
    }
}
```

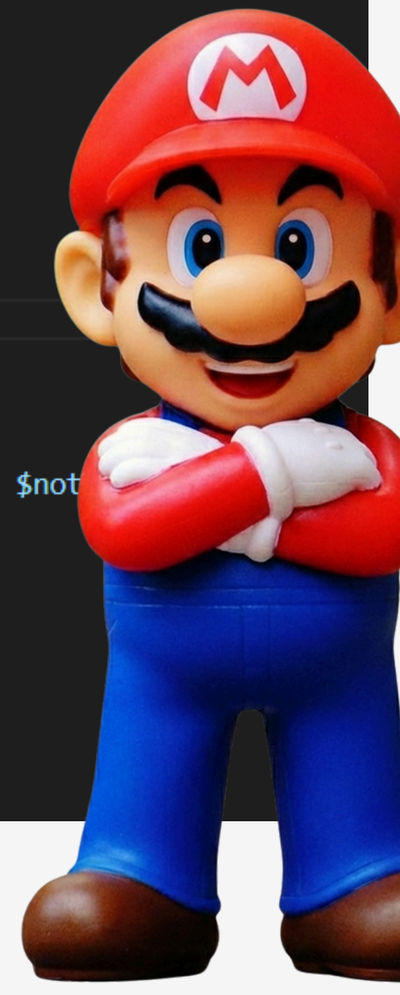
```php
$notifier = new BirdNotifier(new EmailNotifier());
$notifier->notify("Un aigle vient de voler !");

$notifierSms = new BirdNotifier(new SmsNotifier());
$notifierSms->notify("Un pingouin vient de marcher !");
```

DIP

Séparer la logique métier des détails techniques. C'est le principe qui rend l'Injection de Dépendances si puissante.

- ***Découplage***
- ***Flexibilité / extensibilité***
- ***Testabilité***

```php
<?php

interface NotifierInterface {
    public function send(string $message): void;
}

class EmailNotifier implements NotifierInterface {
    public function send(string $message): void {
        echo "Email envoyé : $message";
    }
}

class SmsNotifier implements NotifierInterface {
    public function send(string $message): void {
        echo "SMS envoyé : $message";
    }
}

class BirdNotifier {
    private NotifierInterface $notifier;

    public function __construct(NotifierInterface $not
        $this->notifier = $notifier;
    }

    public function notify(string $msg): void {
        $this->notifier->send($msg);
    }
}
```



Conclusion

Pourquoi les principes SOLID ?

- Maintenabilité
- Extensibilité
- Testabilité
- Lisibilité

MERCI