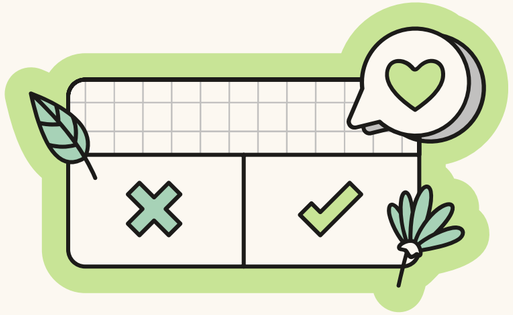
LA MÉTHODE SOLID

By Yanis & Samy



INTRODUCTION

SOLID est un acronyme anglais de 5 principes de la conception orientée objet, qui visent à améliorer l'efficacité du code. Ces principes ont été publiés entre 2000 et 2004 à travers une série d'articles de **Robert C. Martin**, un ingénieur américain.

LES 5 PRINCIPES

Responsabilité unique (**S**ingle responsibility)

Ouvert-fermé (**O**pen-closed)

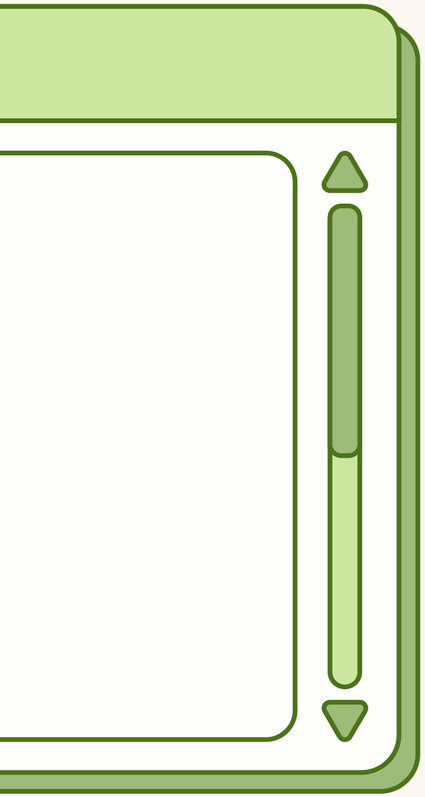
Substitution de Liskov (**L**iskov substitution)

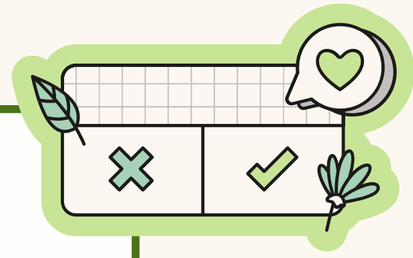
Ségrégation des interfaces (**I**nterface segregation)

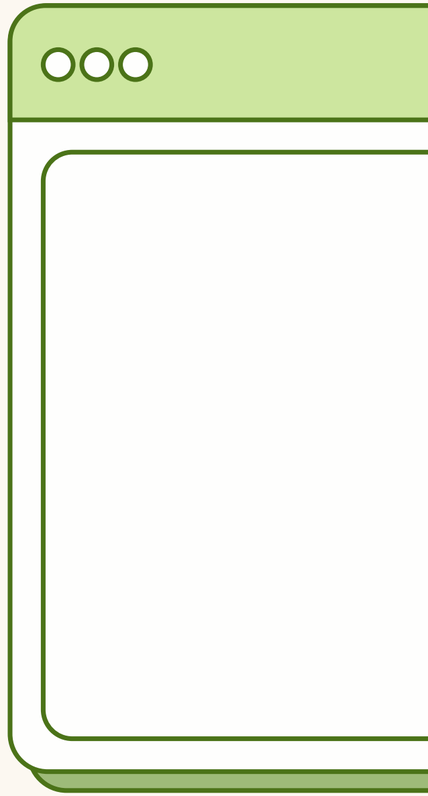
Inversion des dépendances (**D**ependency inversion)

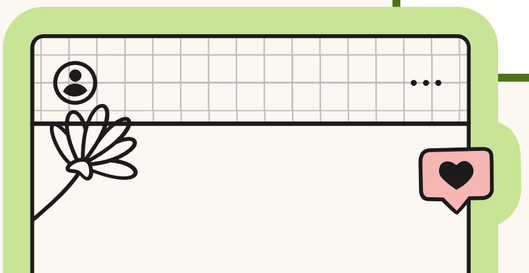
PRINCIPE N°1

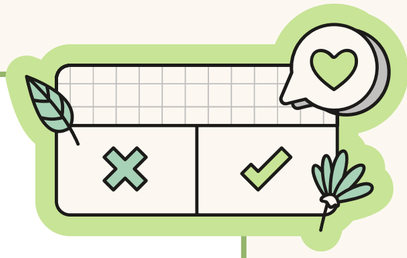
Le principe de la responsabilité unique (**Single responsibility**) stipule qu'une classe ne doit avoir qu'une seule responsabilité. Ceci a pour but de faciliter la maintenance et d'éviter d'avoir des classes surchargées.











EXEMPLE N°1

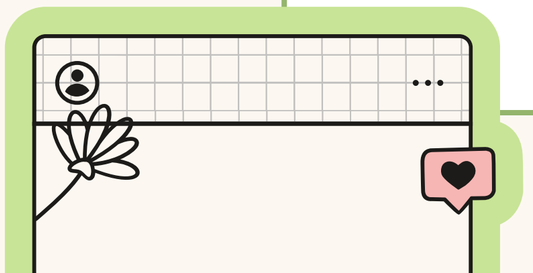
Imaginons que l'on gère des utilisateurs et leur authentification. Voici une mauvaise approche où une classe UserManagement s'occupe de tout :

MAUVAISE APPROCHE

```
class UserManagement {
    public function create($name, $email) {
        // Code pour créer un utilisateur
        $this->sendEmail($email);
    }

    private function sendEmail($email) {
        // Code pour envoyer un email
    }
}
```

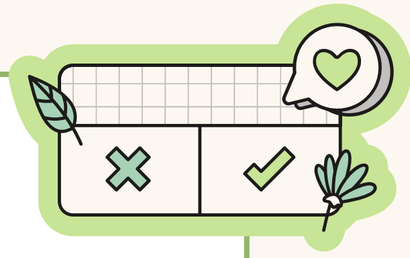
BONNE APPROCHE

```
class User {
    public function createUser($name, $email) {
        // Code pour créer un utilisateur
    }
}

class EmailService {
    public function sendEmail($email) {
        // Code pour envoyer un email
    }
}
```



PRINCIPE N°2

Le principe Ouvert-fermé (**Open-closed**) stipule que les logiciels doivent être ouverts à l'extension mais fermés à la modification. Ainsi, il sera possible d'ajouter de nouvelles fonctionnalités sans changer le code existant.



EXEMPLE N°2

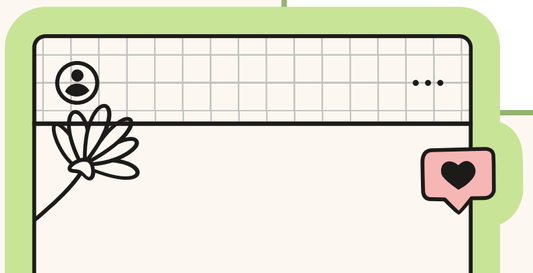
Imaginons que l'on veuille étendre l'envoi d'emails pour supporter également l'envoi de SMS.

MAUVAISE APPROCHE

```
class NotificationService {
    public function sendEmail($email) {
        // Code pour envoyer un email
    }

    public function sendSMS($phoneNumber) {
        // Code pour envoyer un SMS
    }
}
```

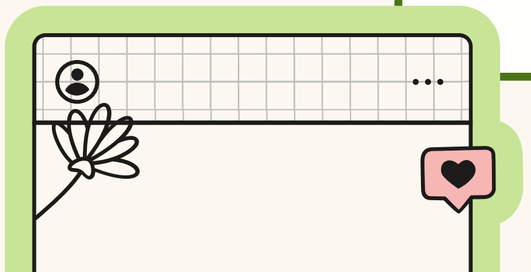
BONNE APPROCHE

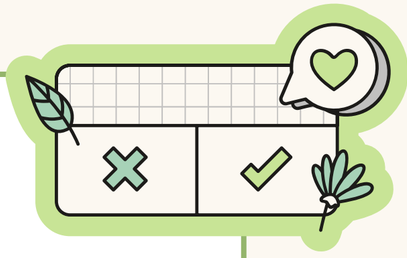
```
interface Notification {
    public function send($recipient);
}

class EmailService implements Notification {
    public function send($email) {
        // Code pour envoyer un email
    }
}

class SMSService implements Notification {
    public function send($phoneNumber) {
        // Code pour envoyer un SMS
    }
}
```



PRINCIPE N°3

Le principe de la substitution de Liskov (**Liskov substitution**) stipule qu'une sous-classe doit pouvoir être utilisée à la place de sa classe mère. Ce qui signifie que les objets d'une classe enfant doivent pouvoir remplacer les objets de la classe parent. Pour cela, chaque attributs/méthodes de la classe parent doit être cohérente avec les classes enfants.





EXEMPLE N°3

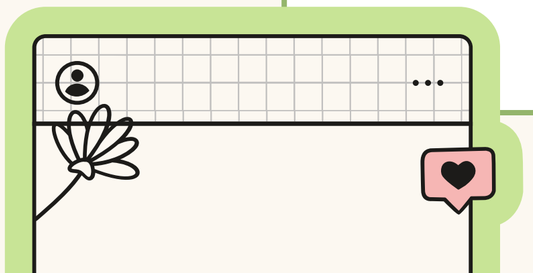
Supposons que nous avons une classe Bird et une classe dérivée Penguin. Le pingouin ne peut pas voler, ce qui ne respecte pas ce principe.

MAUVAISE APPROCHE

```
class Bird {
    public function fly() {
        echo "L'oiseau vole";
    }
}

class Penguin extends Bird {
    public function fly() {
        throw new Exception("Le pingouin ne peut
        pas voler !");
    }
}
```

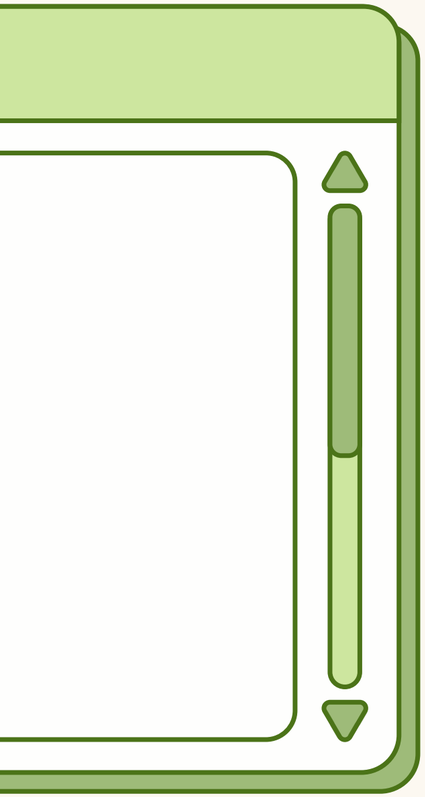
BONNE APPROCHE

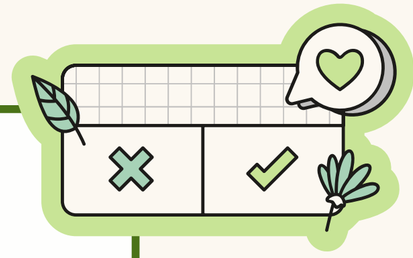
```
class Bird {
    public function move() {
        echo "L'oiseau se déplace";
    }
}
class FlyingBird extends Bird {
    public function move() {
        echo "L'oiseau vole";
    }
}
class Penguin extends Bird {
    public function move() {
        echo "Le pingouin nage";
    }
}
```

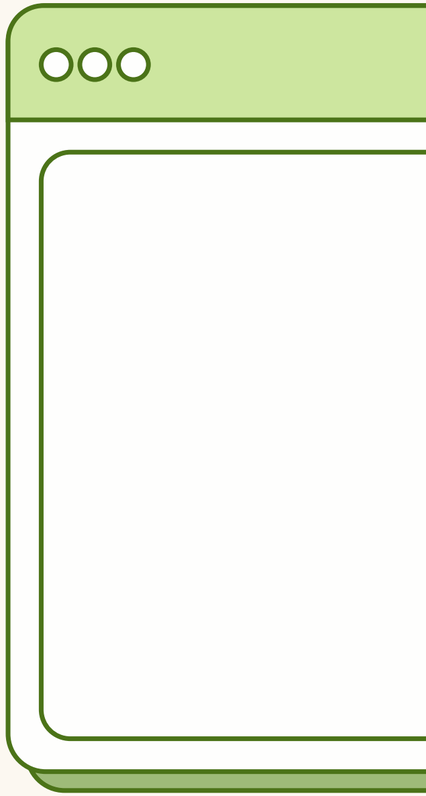


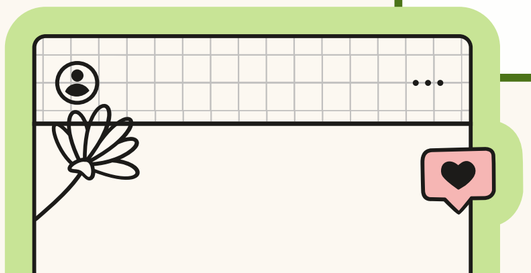
PRINCIPE N°4

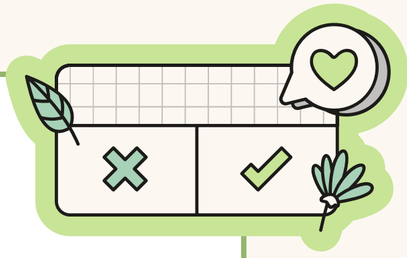
Le principe de la ségrégation des interfaces (**Interface segregation**) stipule que chaque client doit posséder une interface spécifique, afin d'éviter aux classes de dépendre de méthodes dont elles n'ont pas besoin. Privilégier plusieurs petites interfaces plutôt qu'une grande interface générale.











EXEMPLE N°4

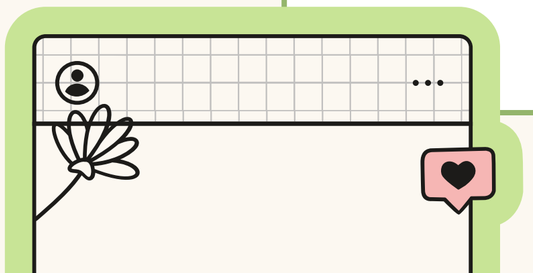
Plutôt que d'avoir une interface AnimalActions qui définit plusieurs méthodes, il faut privilégier des interfaces spécifiques pour chaque comportement.

MAUVAISE APPROCHE

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

class Dog implements AnimalActions {
    public function eat() { echo "Le chien
        mange."; }
    public function fly() { echo "Erreur : un
        chien ne vole pas."; }
    public function swim() { echo "Le chien
        nage."; }
}
```

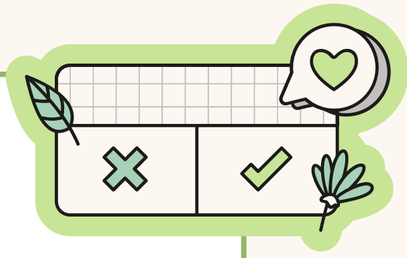
BONNE APPROCHE

```
interface Eatable {
    public function eat();
}
interface Flyable {
    public function fly();
}
interface Swimmable {
    public function swim();
}
class Dog implements Eatable, Swimmable {
    public function eat() { echo "Le chien
        mange."; }
    public function swim() { echo "Le chien
        nage."; }
}
```



PRINCIPE N°5

Le principe d'inversion des dépendances (**Dependency inversion**) stipule que la modification d'une classe de bas niveau ne doit pas entraîner la modification d'une classe de haut niveau. Les classes abstraites ne doivent pas dépendre des classes spécifiques. On évite ainsi les modifications imprévues et/ou indésirables concernant une partie du code.



EXEMPLE N°5

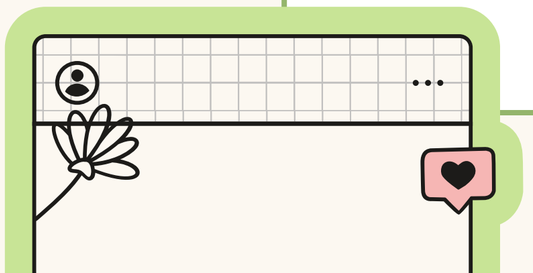
Dans cette application de traitement de paiements, PaymentService est directement couplé à CreditCardPayment. L'ajout d'autre moyen de paiement pousse à modifier PaymentService.

MAUVAISE APPROCHE

```
class CreditCardPayment {
    public function processPayment($amount) {
        // Code pour traiter un paiement par cb
    }
}
class PayPalPayment {
    public function processPayment($amount) {
        // Code pour traiter un paiement PayPal
    }
}
class PaymentService {
    private $creditCardPayment;
    public function __construct() {
       $this->creditCardPayment = new CBPayment();
    }
    public function process($amount) {
       $this->creditCardPayment->processPayment($amount);
    }
}
```

BONNE APPROCHE

```
interface PaymentMethod {
    public function processPayment($amount);
}
class CreditCardPayment implements PaymentMethod {
    public function processPayment($amount) { }
}
class PayPalPayment implements PaymentMethod {
    public function processPayment($amount) { }
}
class PaymentService {
    private $paymentMethod;
    public function __construct(PaymentMethod $paymentMethod) {
        $this->paymentMethod = $paymentMethod;
    }
    public function process($amount) {
        $this->paymentMethod->processPayment($amount);
    }
}
```



THANK YOU