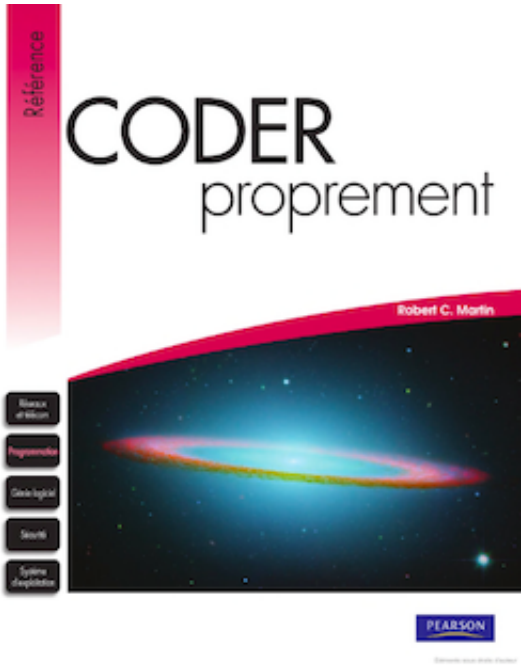
Principes SOLID et design patterns PHP*

Partie 1 : Principes SOLID

[R3.01 : Développement web]

Théorisé en 2002 par Robert C. Martin dans son ouvrage *Agile Software Development, Principles, Patterns and Practices*, l'acronyme SOLID est un moyen mnémotechnique pour retenir 5 **grands principes** applicables au développement d'applications logicielles pour les rendre plus faciles à comprendre, à maintenir et à faire évoluer.



"Coder proprement" - la couverture française du livre de Robert C. Martin
"Agile Software Development, Principles, Patterns and Practices" (tous droits réservés)

Ces 5 principes sont :

- **S** comme **Single Responsibility** Principle ;
- **O** comme **Open/Closed** Principle ;
- **L** comme **Liskov Substitution** Principle ;
- **I** comme **Interface Segregation** Principle ;
- **D** comme **Dependency Inversion** Principle.

*`https://openclassrooms.com`

1 Préparez votre environnement de travail

1.1 Gestion du code avec GitHub

1.1.1 Initialisation d'un dépôt local Git

Vous allez travailler avec Git. Pour ce faire, vous devez créer un dépôt local, c'est à dire un dossier dans lequel toutes vos modifications seront enregistrées.

1. Créez un dossier nommé `solid-php` dans le répertoire de publication web de votre serveur.
2. Placez-vous dans ce dossier `solid-php` nouvellement créé et ouvrez une invite de commande Git Bash ou Windows PowerShell.
3. Lancer la commande `git init`. Elle initialise ce dossier `solid-php` comme un dépôt.

Un dossier caché `.git` a été créé. Il contient tous les éléments non visibles de Git : la configuration, les logs, les branches...

4. Passez la commande `echo "# solid-php" » README.md`.
5. Passez la commande `git add README.md` pour indexer le tout nouveau contenu du dépôt.
6. Utilisez la commande `git commit -m "SOLID and design patterns in PHP : first commit"` pour créer une première version.

L'argument `-m` permet de définir un message particulier rattaché au commit effectué. Si vous n'utilisez pas cet argument, la commande `git commit` vous demandera de saisir le message de commit.

Enfin tapez la commande `git branch -M main`.

1.1.2 Création d'un dépôt distant GitHub

Pour mettre votre projet sur GitHub, vous devez d'abord créer un repository (nommez-le aussi solid-php) dans lequel il pourra être installé.

Puis vous allez devoir relier votre dépôt local au dépôt distant que vous venez de créer sur GitHub.

Pour cela, copiez le lien https (de la forme `https://github.com/alexbrabant/solid-php.git`) qui se trouve en haut sur fond bleu dans l'onglet Code et collez-le dans la commande `git remote add origin https://github.com/alexbrabant/solid-php.git`.

Vous avez relié le dépôt local au dépôt distant. Vous pouvez donc envoyer des commits du repository vers le dépôt distant GitHub en utilisant la commande `git push -u origin main`

Enfin, n'oubliez pas de m'inviter en tant que collaborateur (alexbrabant) ;-)

2 S comme Single Responsibility Principle

Le principe SRP pour "Single Responsibility Principle" implique qu'une classe ne devrait avoir qu'une seule et unique raison de changer.

L'idée ici est de faire en sorte qu'une classe ne soit responsable que d'une seule fonction de votre application, et que cette responsabilité soit complètement encapsulée ("cachée") dans la classe.

L'objectif du principe SRP est de réduire la complexité de votre projet.

Le principe SRP est très simple à mettre en place : il suffit de diviser vos classes en de multiples classes ayant chacune une et une seule responsabilité.

2.1 Exemple

Prenons l'exemple ci-dessous d'une classe `Tools` qui a de multiples responsabilités :

```php
<?php

// src/Tools.php
class Tools
{
    public static function redirectAdmin($url)
    {
        header('Location: ' . $url);
        exit;
    }

    public static function dateFormat($params, &$smarty)
    {
        return Tools::displayDate($params['date'], null,
        (isset($params['full']) ? $params['full'] : false));
    }

    public static function displayDate($date, $id_lang = null,
    $full = false, $separator = null)
    {
        if ($id_lang !== null) {
            Tools::displayParameterAsDeprecated('id_lang');
        }

        if ($separator !== null) {
            Tools::displayParameterAsDeprecated('separator');
        }

        if (!$date || !($time = strtotime($date))) {
```

```
            return $date;
        }

        if ($date == '0000-00-00 00:00:00' || $date == '0000-00-00') {
            return '';
        }

        if (!Validate::isDate($date) || !Validate::isBool($full)) {
            throw new \Exception('Invalid date');
        }

        $context = Context::getContext();
        $date_format = ($full ? $context->language->date_format_full :
        $context->language->date_format_lite);
        return date($date_format, $time);
    }
}
```

Voici la liste des quelques responsabilités du code :

- formatage de dates avec les fonctions `displayDate` et `dateFormat` ;
- action de redirection HTTP ;
- et une fonction qui sert à marquer un paramètre de fonction comme déprécié.

Ayant connaissance de tout cela, nous pouvons donc découper cette classe en 3 classes qui auront chacune un nom beaucoup plus adapté à sa fonction :

- la classe `HttpHeaders` par exemple ne contiendrait que des fonctions qui permettraient de manipuler des en-têtes HTTP ;
- la classe `DateFormatter` aurait la responsabilité de formater des dates au format voulu ;
- enfin, la classe `Deprecator` pourrait contenir toutes les fonctions pour informer le développeur qu'il utilise un code qui n'est plus maintenu, et est voué à disparaître.

Voici ci-après les codes mis à jour qui respectent le premier principe SOLID.

La classe `HttpHeaders` :

```
<?php

// src/HttpHeaders.php
class HttpHeaders
{
    public static function redirect($url) {/*...*/}
}
```

La classe `DateFormatter` :

```php
<?php

// src/Deprecator.php
class Deprecator
{
    public static function displayParameterAsDeprecated($param) {/*...*/}
}
```

La classe `Deprecator` :

```php
<?php

use Deprecator;

// src/DateFormatter.php
class DateFormatter
{
    public static function dateFormat($params, &$smarty) {/*...*/}

    public static function displayDate($date, $id_lang = null, $full = false,
    $separator = null)
    {
        if ($id_lang !== null) {
            Deprecator::displayParameterAsDeprecated('id_lang');
        }

        if ($separator !== null) {
            Deprecator::displayParameterAsDeprecated('separator');
        }

        if (!$date || !($time = strtotime($date))) {
            return $date;
        }

        if ($date == '0000-00-00 00:00:00' || $date == '0000-00-00') {
            return '';
        }

        if (!Validate::isDate($date) || !Validate::isBool($full)) {
            throw new \Exception('Invalid date');
        }

        $context = Context::getContext();
        $date_format = $full ?
            $context->language->date_format_full :
            $context->language->date_format_lite
        ;
```

```
        return date($date_format, $time);
    }
}
```

Le principe SRP est aussi applicable aux fonctions. Dès le moment où une fonction a trop de responsabilités, n'hésitez pas à réduire sa complexité en la découpant en plusiseurs fonctions plus simples, et qui ont une seule et unique responsabilité.

En particulier dans notre exemple, nous pouvons aller plus loin pour la classe `DateFormatter` : la fonction `displayDate` a beaucoup trop de responsabilités, dont une bonne partie a été dépréciée ! Voici donc une version améliorée de cette classe :

```
<?php

use Deprecator;
use DateValidator;

// src/DateFormatter.php
class DateFormatter
{
    public static function dateFormat($params, &$smarty) {/*...*/}

    public static function displayLite($date)
    {
        $time = DateValidator::validate($date);
        $date_format = Context::getContext()->language->date_format_lite;
        return date($date_format, $time);
    }

    public static function displayFull($date)
    {
        $time = DateValidator::validate($date);
        $date_format = Context::getContext()->language->date_format_lite;
    }

    // fonction dépréciée, utilisez plutôt displayLite ou displayFull !
    public static function displayDate($date, $id_lang = null, $full = false,
    $separator = null)
    {
        if ($id_lang !== null) {
            Deprecator::displayParameterAsDeprecated('id_lang');
        }

        if ($separator !== null) {
            Deprecator::displayParameterAsDeprecated('separator');
        }
```

```
        if (!$date || !($time = strtotime($date))) {
            return $date;
        }

        if ($date == '0000-00-00 00:00:00' || $date == '0000-00-00') {
            return '';
        }

        if (!Validate::isDate($date) || !Validate::isBool($full)) {
            throw new \Exception('Invalid date');
        }

        $context = Context::getContext();
        $date_format = $full ?
            $context->language->date_format_full :
            $context->language->date_format_lite
        ;

        return date($date_format, $time);
    }
}

class DateValidator
{
    public static function validate($date)
    {
        if (!$date || !($time = strtotime($date))) {
            return $date;
        }

        if ($date == '0000-00-00 00:00:00' || $date == '0000-00-00') {
            return '';
        }

        if (!Validate::isDate($date) || !Validate::isBool($full)) {
            throw new \Exception('Invalid date');
        }

        return $time;
    }
}
```

2.2 Travail à faire

Déposez dans votre répertoire de travail `solid-php` le dossier `exo-solid-1`.

Dans ce répertoire `exo-solid-1`, le fichier `Uploader.php` contient une classe `Uploader` qui a plusieurs responsabilités :

- le téléchargement de fichiers, bien sûr ;
- le redimensionnement d'images ;
- la vérification du type MIME des fichiers ;
- la récupération de l'extension des fichiers.

Commencez par créer les différentes classes. Puis remplacez le code de chaque fonction qui n'est pas à sa place dans la classe `Uploader` par un appel à la fonction de la nouvelle classe.
Par exemple :

```php
<?php

use FileInformation;

class Uploader
{
    public function getExtension()
    {
        $fileInformation = new FileInformation();

        return $fileInformation->getExtension($this->name);
    }
}
```

À ce stade, le code devrait toujours fonctionner sans problème : après tout, vous avez seulement déplacé le code.

Ensuite, recherchez dans le script de test (`app.php`) de l'application les appels aux fonctions de la classe `Uploader` et remplacez-les (lorsque c'est nécessaire) par des appels à vos nouvelles classes : votre application devrait toujours fonctionner.

Enfin, vérifiez le comportement en exécutant le script de test `app.php` : si le code ne fonctionne plus, c'est que vous avez oublié de remplacer certains appels de fonction.

2.2.1 Validation

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
```

```
> git commit -m "S as Single Responsibility Principle"
> git push -u origin main
```

3 O comme Open/Closed Principle

Le principe OCP pour "Open/Closed Principle" signifie que les classes d'un projet devraient être ouvertes à l'extension, mais fermées à la modification.

L'idée de fond derrière ce principe est la suivante : ajouter de nouvelles fonctionnalités ne devrait pas casser les fonctionnalités déjà existantes.

Une classe est considérée comme ouverte s'il est possible de l'étendre et d'en changer le comportement.

En PHP, si une classe n'est pas `final`, vous pouvez l'étendre et donc hériter de ses propriétés, fonctions et constantes publiques.

```php
<?php

class A
{
    public function helloWorld()
    {
        return 'Hello World';
    }
}

class B extends A
{
    // vide
}

$b = new B();

$b->helloWorld(); // 'Hello World'
```

Si par contre, une classe est déclarée `final` (on dit également "complète"), il n'est pas possible de l'étendre, et une erreur fatale sera levée par PHP.

```php
<?php

final class A {}

class B extends A {}

// PHP Fatal error:  Class B may not inherit from final class (A)
```

Enfin, il existe un type de classe considéré comme "incomplet" par défaut : il s'agit des classes `abstract` (abstraites).
Une classe `abstract` ne peut pas être instanciée directement, elle ne peut qu'être héritée.
De plus, on peut définir des fonctions abstraites qu'il faudra obligatoirement implémenter dans les classes dites "enfants" :

```php
<?php

abstract class A
{
    public function helloWorld()
    {
        return 'Hello World';
    }

    abstract public function warning();
}

$a = new A();

// PHP Warning:  Uncaught Error: Cannot instantiate abstract class A

class B extends A
{
    // vide
}

// PHP Fatal error:  Class B contains 1 abstract method and must
// therefore be declared abstract or implement the remaining methods (A::warning)
```

Il est logiquement impossible d'avoir une classe qui `final` et `abstract` en même temps.

Conclusion : plutôt que de changer le comportement d'une classe existante pour l'adapter à un nouveau besoin, il vaut mieux étendre cette classe et en adapter le comportement : elle est donc ouverte à l'extension et fermée à la modification.

On évite ainsi de casser ou d'introduire des bugs dans une application qui fonctionne correctement.

3.1 Exemple

Imaginons une classe qui doit calculer les coûts de transport d'une commande lors d'un achat sur une plateforme e-commerce. Ce coût peut dépendre du moyen de transport, du mode de livraison (l'option "rapide" ou au contraire "économique"), et aussi du poids des produits de la commande.

Voici à quoi la classe pourrait ressembler :

```php
<?php

class Order
{
    // eco ou rapide
    private $shippingPlan;

    // en kg
    private $orderWeight;

    // avion ou train par exemple
    private $shippingMode;
    /*...*/

    public function getShippingCost()
    {
        $cost = '10';

        if ($this->shippingMode == 'avion') {
            $cost = $this->orderWeight * 3;

            if ($this->shippingPlan == 'rapide') {
                $cost = 1.3 * $cost;
            }
        }

        if ($this->shippingMode == 'train') {
            $cost = $this->orderWeight * 2;

            if ($this->shippingPlan == 'rapide') {
                $cost = 1.1 * $cost;
            }
        }

        return $cost;
    }
}
```

Si nous devions rajouter un autre mode de livraison, ou alors un tarif différent selon le nom du transporteur ou encore selon le pays, le code de la fonction `getShippingCost` deviendrait complexe.

Nous allons donc rendre ce code compatible avec le principe OCP, en utilisant le polymorphisme. Le polymorphisme, c'est la capacité d'un langage à supporter différentes implémentations d'un même système.
L'héritage de classe est une forme de polymorphisme.

Le coût d'un envoi dépend du mode d'envoi, du poids et du véhicule utilisé...

Ce n'est pas de la responsabilité de la classe `Order` de calculer tout ça.

Créer un objet `ShippingType` qui prend une commande en paramètre permet de :

- séparer les responsabilités (SRP) ;
- pouvoir créer un type de `ShippingType` par véhicule (OCP).

Voici le code mis à jour pour respecter les deux principes SOLID abordés :

```php
<?php
use Order;

abstract class ShippingType
{
    abstract public function getCost(Order $order);
}

class AircraftShipping extends ShippingType
{
    public function getCost(Order)
    {
        /*...*/
    }
}

class TrainShipping extends ShippingType
{
    public function getCost(Order)
    {
        /*...*/
    }
}

class Order
{
    private $shippingType;

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

    public function getShippingCost()
    {
        return $this->shippingType->getCost();
    }
}
```

Détaillons les modifications :

Tout d'abord, nous avons créé une classe abstraite `ShippingType` avec la méthode abstraite `getCost(Order)`.

En faisant cela, pour implémenter une nouvelle méthode de livraison, il nous suffira de créer une nouvelle classe qui étend `ShippingType`.

Livraison par drone ?

```php
<?php

use Order;

class DroneShipping extends ShippingType
{
    public function getCost(Order $order)
    {
        /*...*/
    }
}
```

Il suffit ensuite de sélectionner la bonne méthode d'envoi pour que la classe `Order` soit "capable" de calculer le coût d'envoi de la commande.
Pour cela, nous avons décidé d'injecter la classe correspondante en constructeur.
En faisant cela, nous déléguons la responsabilité de ce calcul à l'implémentation de la classe `ShippingType`.

La classe `Order` ne changera donc plus quand il faudra ajouter ou modifier une règle de calcul, ce qui était notre objectif pour respecter ce principe.

3.2 Travail à faire

Déposez dans votre répertoire de travail `solid-php` le dossier `exo-solid-2`.

Dans ce projet, vous remarquerez qu'il existe une classe appelée `Music`. L'idée originale de l'auteur de ce projet était que l'application puisse supporter différents types de formats audio comme le MP3 ou le Ogg, par exemple.
Malheureusement, le code existant a quelques problèmes de conception que vous êtes maintenant capable de corriger.
Améliorez le code des classes `Music` et `MP3` pour qu'elles respectent le principe Open/Closed.

3.2.1 Validation

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
> git commit -m "O as Open/Closed Principle"
> git push -u origin main
```

4 L comme Liskov Substitution Principle

Le troisième principe SOLID met en valeur non pas une, mais deux brillantes développeuses !
En effet, le principe de substitution a été formulé par Barbara Liskov et Jeannette Wing dans un article intitulé Family Values: A Behavioral Notion of Subtyping [EN] en 1994.
Si le principe est nommé d'après Barbara uniquement, c'est parce qu'elle l'avait une première fois défini dans un article intitulé Data abstraction en hierarchy [EN] en 1987.

Ce principe complète le second principe : il doit être possible de substituer une classe "parente" par l'une de ses classes enfants (on dit aussi "dérivées").
Pour cela, nous devons garantir que les classes enfants auront le même comportement que la classe qu'elles étendent.

Dans l'exemple précédent, si nous avions eu des classes enfants qui renvoient des entiers, et d'autres des chaînes de caractères ("20 €"), nous aurions donc changé le comportement de la fonction `Order::getShippingCost`, et probablement introduit un bug dans notre application.

Pour garantir que l'on ne va pas tout casser, le principe de substitution de Liskov donne une liste de règles strictes. Nous allons en présenter les principales.

4.1 Contrôlez les types passés en paramètres de méthodes

Les types de paramètres dans la méthode d'une classe enfant doivent correspondre ou être plus "abstraits" que les types correspondants dans la classe parente.

Exemple :

```php
<?php

class ParentOrder {}
class Order extends ParentOrder {}

class SubOrder extends Order {}

class Cart
{
    public function getShippingCost(Order $order)
```

```
    {
        /*...*/
    }
}

// Mauvaise idée!
class BadSubCart extends Cart
{
    public function getShippingCost(SubOrder $order)
    {
        /*...*/
    }
}

// Bonne idée!
class GoodSubCart extends Cart
{
    public function getShippingCost(ParentOrder $order)
    {
        /*...*/
    }
}
```

Nous sommes dans la classe `Cart` ("panier" d'un site de e-commerce), et nous cherchons toujours à calculer les coûts de livraison.

Pourquoi risquons-nous de créer un bug en remplaçant la classe `Cart` par `BadSubCart` ?

Pour que la classe SubCart puisse se substituer sans risque à la classe Cart, il faut que le paramètre passé à la méthode `getShippingCost` soit :

- une instance de `Order` ;
- ou une instance parente de `Order` (qui peut le plus, peut le moins).

4.2 Contrôlez les types passés en retour de méthodes

Le type de retour d'une méthode d'une classe enfant doit correspondre, ou être un "sous-type" du type de retour de la classe parente.

Comme vous pouvez le voir, les prérequis sont cette fois contraires à la règle précédente. Quand on y réfléchit, c'est assez logique : si une classe est capable de gérer une fonction qui retourne un `Order`, elle devrait être capable de gérer un `SubOrder`.

Exemple :

```php
<?php

class ParentOrder {}
class Order extends ParentOrder {}
class SubOrder extends Order {}

class Cart
{
    public function getOrder()
    {
        /*...*/

        return new Order();
    }
}

// Bonne idée!
class GoodSubCart extends Cart
{
    public function getOrder()
    {
        /*...*/

        return new ParentOrder();
    }
}

// Mauvaise idée!
class BadSubCart extends Cart
{
    public function getOrder()
    {
        /*...*/

        return new SubOrder();
    }
}
```

Il existe un moyen de forcer le code à respecter certaines de ces règles : l'interface (que l'on appelle parfois "contrat") en programmation orientée objet joue parfaitement ce rôle.

4.3 Tirez profit des interfaces en programmation orientée objet

Une interface va vous permettre de définir proprement le contrat que doivent respecter les objets qui l'implémentent.
De cette façon, il n'est plus possible de se tromper sur les types d'entrée ou de retour de vos fonctions !

L'essentiel des langages de programmation sont capables de contractualiser ces contraintes : pour PHP, c'est depuis la version 7.

```php
<?php

class ShippingType {}
class SubShippingType extends ShippingType {}

class ParentOrder {}
class Order extends ParentOrder {}
class SubOrder extends Order {}

interface Cart
{
    public function getOrder(ShippingType $type) : Order;
}

// Fatal error: Declaration of WrongParamCart::getOrder(SubShippingType $type):
// Order must be compatible with Cart::getOrder(ShippingType $type): Order
class WrongParamCart implements Cart
{
    public function getOrder(SubShippingType $type) : Order
    {
        return new Order();
    }
}

// Fatal error: Uncaught TypeError: Return value of WrongReturnCart::getOrder()
// must be an instance of Order, instance of ParentOrder returned
class WrongReturnCart implements Cart
{
    public function getOrder(ShippingType $type) : Order
    {
        return new ParentOrder();
    }
}

// Pas d'erreur ici, les contraintes de l'interface sont respectées
class RightCart implements Cart
{
    public function getOrder(ShippingType $type) : Order
    {
        return new Order();
    }
}
```

Si vous utilisez des interfaces, vous n'aurez plus à vous inquiéter du principe de Liskov : c'est votre

langage de programmation qui s'en préoccupera pour vous.
À noter que le résultat aurait été le même avec une classe abstraite `Cart` plutôt qu'une interface, le principal étant de bien typer vos paramètres et vos méthodes, comme c'est le cas ici :
`public function getOrder(ShippingType $type) : Order`
Nous avons précisé que `getOrder` prend en paramètre un objet `ShippingType` et retourne un objet `Order`.

4.4 Contrôlez les types d'exceptions lancées par les fonctions

Comme pour les types de retours de fonction, cette règle stipule que si une exception est lancée par une classe parente dans certaines conditions, la classe enfant doit également lancer une exception dans ces conditions, et cette exception doit être de même type ou sous-type que l'exception parente. C'est une règle parfaitement logique.
En effet, si on s'attend à attraper une exception de type `InvalidShippingCostException` et qu'on remplace notre implémentation par une autre, et que cette fois l'exception n'est ni une `InvalidShippingCostException` ni l'une de ses classes dérivées, alors nous ne serons pas en capacité de l'attraper.

```php
<?php

class InvalidShippingCostException extends \Exception {}
class BadFormattedCostException extends InvalidShippingCostException {}
class WrongShippingMethodException extends \Exception {}

try {
    throw new InvalidShippingCostException();
} catch (InvalidShippingCostException $e) {
    echo 'ERREUR!';
}

try {
    throw new BadFormattedCostException();
} catch (InvalidShippingCostException $e) {
    echo 'ERREUR 2!';
}

try {
    throw new WrongShippingMethodException();
} catch (InvalidShippingCostException $e) {
    echo 'ERREUR 3!';
}

// ERREUR!ERREUR 2!
```

Il faut donc être vigilant sur le type d'exception utilisé si l'on ne souhaite pas "casser" le code dont va dépendre le comportement de notre application.

Les langages compilés comme C#, Java ou C++ sont capables de détecter ce type d'erreur, mais pas le langage PHP.

4.5 Exemple

Reprenons une nouvelle fois l'exemple des coûts de livraison et commençons par définir l'interface pour le type de livraison :

```php
<?php

use Order;
use InvalidShippingCost;

interface ShippingType
{
    /**
     * @throws InvalidShippingCost
     */
    public function getCost(Order $order) : float;
}
```

Nous n'avons donc plus besoin de la classe abstraite (elle tenait le rôle de "contrat léger" dans l'exemple précédent).

Maintenant, nous avons le contrôle total sur le type d'entrée (un `Order`) et le type de retour (une valeur flottante), mais que se passe-t-il en cas d'erreur ? Nous devons lever une exception.

```php
<?php

use Exception;

abstract class InvalidShippingCost extends Exception
{
}
```

Et maintenant, chaque implémentation de `ShippingType` doit implémenter l'interface et lancer une exception de type `InvalidShippingCost`.

Améliorons l'implémentation pour le transport par drone :

```php
<?php

class TooHeavyOrderException extends InvalidShippingCost {}
```

```php
class DroneShippingType implements ShippingType
{
    public function getCost(Order $order) : float
    {
        if ($order->getWeight() > 10) {
            throw new TooHeavyOrderException();
        }

        $cost = $order->getWeight() * 3;

        if ($order->getPlan() == 'rapide') {
            $cost = 1.3 * $cost;
        }

        return (float) $cost;
    }
}
```

Chaque classe doit être dans son propre fichier. Ici elles sont rassemblées uniquement pour des raisons pratiques.

4.6 Travail à faire

Vous pouvez encore améliorer le projet de lecteur de musiques extrait du projet fil rouge car il n'y a pas encore de gestion d'erreurs.

Déposez dans votre répertoire de travail `solid-php` le dossier `exo-solid-3`.

Mettez en place les interfaces adéquates pour les classes `MP3` et `Ogg` et assurez-vous que si on passe le mauvais type de fichier au lecteur, une exception est lancée depuis la classe parente `InvalidFileException` :

- l'exception retournée sera de type `InvalidExtensionException` si l'on essaie de lire un fichier Ogg avec la classe `MP3` (et vice versa) ;
- l'exception retournée sera de type `UnknownExtensionException` si l'on essaie de lire un fichier sans aucune extension.

4.6.1 Validation

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
> git commit -m "L as Liskov Substitution Principle"
> git push -u origin main
```

5 I comme Interface Segregation Principle

Le quatrième principe SOLID commence donc par la lettre I pour "Interface Segregation Principle".
Nous pourrions le traduire de cette façon : "Vous ne devriez pas avoir à implémenter des méthodes dont vous n'avez pas besoin."
L'objectif ici est de découper une interface aux multiples responsabilités en de multiples interfaces qui ont, elles, une seule responsabilité.
Une responsabilité peut se traduire par l'implémentation de plusieurs méthodes, il ne faut pas non plus créer une interface par méthode publique !
Vous l'aurez compris, ce principe revisite le premier principe (Single Responsibility Principle), en l'appliquant cette fois aux interfaces.

5.1 Exemple

Nous allons changer d'exemple. Cependant nous choisissons à nouveau un exemple adapté d'un projet open source connu, car même du code éprouvé par de nombreux développeurs peut toujours être amélioré. Voici l'interface en question :

```php
<?php

interface UrlFileCheckerInterface
{
    /**
     * Check if .htaccess file for Front Office is writable.
     *
     * @return bool
     */
    public function isHtaccessFileWritable() : bool;

    /**
     * Check if robots.txt file for Front Office is writable.
     *
     * @return bool
     */
    public function isRobotsFileWritable() : bool;
}
```

Détaillons ce code ensemble : Dans les deux cas, nous vérifions qu'un fichier est accessible en écriture.
Mais en quoi vérifier que le fichier `.htaccess` est accessible en écriture a un rapport avec les accès du fichier `robots.txt` ?
Et imaginons que sur un projet, je veuille réutiliser cette interface, mais que je n'aie pas de fichier `.htaccess`, car le projet utilise le logiciel serveur Nginx plutôt que le logiciel Apache.
Je vais alors devoir implémenter une fonction qui sera inutile au projet.
Voilà comment nous pourrions améliorer cela :

```php
<?php

interface IsHtaccessFileWritableInterface
{
    /**
     * Check if .htaccess file for Front Office is writable.
     *
     * @return bool
     */
    public function isWritable();
}

interface IsRobotsFileWritableInterface
{
    /**
     * Check if robots.txt file for Front Office is writable.
     *
     * @return bool
     */
    public function isWritable();
}
```

Mais, quand on y réfléchit... avons-nous réellement besoin d'une interface par fichier ? Améliorons encore cela.

```php
<?php

interface FileWritableInterface
{
    /**
     * Check if a file is writable.
     *
     * @return bool
     */
    public function isWritable();
}

class IsHtaccessFileWritable implements FileWritableInterface
{
```

```
    /**
     * {@inheritdoc}
     */
    public function isWritable()
    {
        /*...*/
    }
}

class IsRobotsFileWritable implements FileWritableInterface
{
    /**
     * {@inheritdoc}
     */
    public function isWritable()
    {
        /*...*/
    }
}
```

Quels principes SOLID (déjà vus) ont été appliqué dans ce code ?

☞ Réponse [1]

5.2 Travail à faire

Déposez dans votre répertoire de travail `solid-php` le dossier `exo-solid-4`.

Cet exercice est fortement inspiré d'une librairie open source que vous pouvez consulter dans ce GitHub.

Cet exercice ne contient qu'une seule interface et deux implémentations.
À l'aide des commentaires de l'interface et du code des deux classes, créez deux interfaces, de sorte que les classes n'aient plus à implémenter de code inutile.
Vous en profiterez pour adapter le code des classes pour qu'elles utilisent vos nouvelles interfaces, au besoin.

5.2.1 Validation

1. Open/Closed Principle et Liskov Substitution Principle

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
> git commit -m "I as Interface Segregation Principle"
> git push -u origin main
```

6 D comme Dependency Inversion Principle

Le cinquième et dernier principe SOLID est le principe d'inversion de dépendances : "Dependency Inversion Principle".
Comprenez : les classes de haut niveau ne devraient pas dépendre directement des classes de bas niveau, mais d'abstractions.
Détaillons ce principe. Tout d'abord, un peu de vocabulaire, quand on conçoit un logiciel en programmation orientée objet, on peut faire la distinction entre deux types de classes :

- Les classes de bas niveau qui implémentent des fonctionnalités de base : écrire dans un fichier, se connecter à une base de données, retourner une réponse HTTP...
- Les classes de haut niveau qui concernent le métier de vos applications (la logique métier) : la gestion des coûts de livraison, par exemple.

Puisque vos classes de haut niveau auront besoin de classes bas niveau, vous aurez commencé par écrire les classes de bas niveau en premier. Le problème de cette approche est que vos classes de haut niveau ont tendance à dépendre directement des classes de bas niveau.

6.1 Exemple

Prenons l'exemple d'une classe `UserRepository` dont la responsabilité est de récupérer une liste d'utilisateurs. Pour cela, nous avons développé une classe `Database` capable de faire des requêtes à la base de données.

```php
<?php

use Database;
use User;

class UserRepository
{
    private $database;

    public function __contruct(Database $database)
    {
        $this->database = $database;
    }
```

```
    public function getAll()
    {
        $users = [];
        $results = $this->database
            ->execute('SELECT * FROM users')
            ->fetchAll()
        ;

        foreach($results as $result) {
            $users[] = new User($result);
        }

        return $users;
    }
}
```

Dans cet exemple, la classe de haut niveau (`UserRepository`) dépend directement d'une classe de bas niveau (`Database`).
C'est problématique, car si un jour nous voulons utiliser autre chose qu'une base de données pour retrouver les utilisateurs (une API REST par exemple), nous serons obligés de changer le code de cette classe (ce qui est d'ailleurs incompatible avec le principe "Open Closed Principle").

La solution consiste à inverser le sens de la dépendance, à faire en sorte que ce soient les classes de bas niveau qui dépendent d'abstractions (classes abstraites et interfaces).
Nous allons donc créer une interface de haut niveau que nos classes de bas niveau devront implémenter.

Commençons par définir une nouvelle interface indiquant qu'une classe de type `DatabaseInterface` doit au moins permettre d'exécuter une requête et de récupérer des données :

```
<?php

interface DatabaseInterface
{
    public function execute($query) : self;

    public function fetchAll() : array;
}
```

Maintenant, notre classe bas niveau peut implémenter l'interface (nous allons la renommer au passage) :

```
<?php

use DatabaseInterface;

class MySQL implements DatabaseInterface
{
```

```
    public function execute($query)
    {
        /*...*/
    }

    public function fetchAll()
    {
         /*...*/
    }
}
```

Et voici le code de la classe `UserRepository` mis à jour :

```
<?php

use DatabaseInterface;
use User;

class UserRepository
{
    private $database;

    public function __contruct(DatabaseInterface $database)
    {
        $this->database = $database;
    }

    public function getAll()
    {
        $users = [];
        $results = $this->database
            ->execute('SELECT * FROM users')
            ->fetchAll()
        ;

        foreach($results as $result) {
            $users[] = new User($result);
        }

        return $users;
    }
}

$userMySQLRepository = new UserRepository(new MySQL());
$userRestApiRepository = new UserRepository(new RestApiClient());
```

Conclusion : avant, c'était la classe de haut niveau qui devait respecter l'implémentation des fonc-

tions de bas niveau. Et maintenant, ce sont les classes de bas niveau qui vont devoir respecter les contraintes des classes de haut niveau.
Comme vous le voyez dans l'exemple mis à jour, cela nous permet d'imaginer le support de plusieurs systèmes de persistance des données.

6.2 Travail à faire

Déposez dans votre répertoire de travail `solid-php` le dossier `exo-solid-5`.

Sur le modèle de l'exemple décrit précédemment, vous allez compléter un programme de récupération d'utilisateurs dans différents systèmes de données :

- des utilisateurs stockés dans des tableaux ;
- des utilisateurs stockés sous forme de fichiers JSON.

Un script de test `app.php` est disponible dans le dossier de l'exercice, et le code à compléter est identifié par des commentaires.

6.2.1 Validation

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
> git commit -m "D as Dependency Inversion Principle"
> git push -u origin main
```