

# Programmation orientée objet en PHP

## Partie 3 : Agencez votre code

Grégoire Hébert

Mis à jour le 14/06/2024

### 1 Spécialisez vos objets avec les espaces de noms

#### 1.1 Découvrez les espaces de noms

Une des règles à garder en tête lorsque l'on conçoit des classes, c'est de limiter leur rayon d'action. **Une classe ne devrait posséder qu'une seule responsabilité** (☞ principes **SOLID**). Il en va de même pour les méthodes qui la composent. Ça évite d'avoir des méthodes trop complexes, d'introduire des bugs, c'est plus facile de les tester et de les faire évoluer.

Du coup, parfois il se trouve que deux classes utilisées pour une action similaire et différente à la fois existent dans votre code.

Par exemple, une classe **Message** pour un forum est à la fois similaire et différente d'une classe **Message** pour une messagerie interne. Ce sont deux classes différentes portant le même nom. En PHP, (comme en C#), il est interdit d'avoir deux classes portant le même nom.



| Pour vous en convaincre, testez le code ci-après.

```
<?php

declare(strict_types=1);

class Message
{}

class Message
{}
```

Le code ci-dessus vous a donc donné une erreur. La correction facile serait de renommer une des deux classes mais sachez qu'il n'y a rien de pire à faire !

On va encapsuler nos classes dans des "espaces" réservés. On encapsule dans l'espace **Forum** pour la première, et l'espace **Messenger** pour la seconde.

Ces espaces concernent surtout le nom de nos classes, c'est pour ça que ça s'appelle les espaces de noms. Ils permettent de regrouper des classes sous un même nom, et surtout d'empêcher les ambiguïtés.

Un parallèle qui peut-être fait avec les espaces de nom (appelés aussi **namespace**) est l'utilisation de dossiers. Vous ne pouvez pas avoir deux fichiers qui ont le même nom (ici nos classes), sauf s'ils sont dans deux dossiers séparés. C'est la même chose ici, nous ne pouvons pas avoir deux classes Message, sauf si elles sont dans deux espaces de noms différents.

## 1.2 Utilisez les espaces de noms

### 1.2.1 Syntaxe utilisant un point-virgule

Les espaces de noms existent dans plusieurs langages. En PHP, il faut utiliser le mot clé **namespace** suivi du nom de cet espace et d'un point-virgule :

```
<?php

declare(strict_types=1);

namespace Forum;
class Message
{}

namespace Messenger;
class Message
{}
```

Toutes les classes déclarées sous le **namespace** seront considérées comme lui appartenant. Dès qu'un nouveau namespace est déclaré, les classes suivantes seront dans ce dernier.

Il existe une syntaxe différente : Au lieu d'un ;, vous pouvez encapsuler l'espace de noms dans des accolades {}. Lorsque vos namespaces sont dans le même fichier, c'est un peu plus lisible. Nous y reviendrons un peu plus loin dans cette section.

Pour faire référence à une classe en particulier, vous devez préfixer son nom par son espace de noms.



| Testez le code ci-après.

```
<?php

declare(strict_types=1);

namespace Forum;
```

```

class Message
{}

namespace Messenger;
class Message
{}

$forumMessage = new \Forum\Message;
$messengerMessage = new \Messenger\Message;

var_dump($forumMessage::class);
var_dump($messengerMessage::class);

```



Vous avez constaté que le namespace est écrit avec des \ comme séparateurs. Il y en a un devant aussi. Par défaut, PHP possède un espace de nom global représenté par cet anti-slash. Vous l'utilisiez jusqu'à présent sans le savoir. Si vous ne précisez aucun namespace, PHP considère que vous faites référence au namespace global.



À partir du moment où une classe fait partie d'un espace nommé, vous ne pouvez plus y faire référence uniquement par son nom, vous êtes obligé d'y faire référence avec son espace de noms. En anglais le sigle FQCN est utilisé, pour *Fully Qualified Class Name* (nom de classe entièrement qualifié).

La fonction `get_class` et la constante de classe `class` retournent le FQCN.

Tous les moyens de comparaison d'objets en PHP prennent en compte l'espace de noms.



Après ce que vous venez de lire, à votre avis, que fait PHP si on crée une instance de `DateTime` à la suite des `Message` dans le code-ci-après ?

```

<?php

declare(strict_types=1);

namespace Forum;

class Message
{}

namespace Messenger;
class Message

```

```

{}

$forumMessage = new \Forum\Message;
$messengerMessage = new \Messenger\Message;
$date = new DateTime();

var_dump($forumMessage::class);
var_dump($messengerMessage::class);

```



PHP pense qu'on est dans l'espace de nom **Messenger**. Or, **DateTime** se situe toujours dans l'espace global. Pour y faire référence, on doit rajouter l'anti-slash devant :

```
$date = new \DateTime();
```

### 1.2.2 Syntaxe utilisant des accolades

Il y a une autre technique pour déclarer les classes dans les espaces de noms, c'est d'encapsuler les espaces de noms avec les accolades. Mais alors se pose une autre contrainte : lorsqu'un espace de noms est déclaré dans un fichier, tout le code de ce fichier doit faire partie d'un espace de noms.

Ainsi, avec les accolades, le code ci-après ne marchera pas car vous devez encapsuler les lignes 15 à 19 dans un namespace.

```

1  <?php
2
3  declare(strict_types=1);
4
5  namespace Forum {
6      class Message
7      {}
8  }
9
10 namespace Messenger {
11     class Message
12     {}
13 }
14
15 $forumMessage = new \Forum\Message;
16 $messengerMessage = new \Messenger\Message;
17
18 var_dump($forumMessage::class);
19 var_dump($messengerMessage::class);

```

S'il s'agit du namespace global, alors il suffit de ne pas nommer l'espace de noms :

```

1  <?php
2
3  declare(strict_types=1);
4
5  namespace Forum {
6      class Message
7      {}
8  }
9
10 namespace Messenger {
11     class Message
12     {}
13 }
14
15 namespace {
16     $forumMessage = new Forum\Message;
17     $messengerMessage = new Messenger\Message;
18
19     var_dump($forumMessage::class);
20     var_dump($messengerMessage::class);
21 }

```

### 1.2.3 Espaces de noms à plusieurs niveaux

Dans l'exemple précédent, il y a un seul niveau, mais on peut avoir des espaces de noms à plusieurs niveaux ! Par exemple :

```

<?php

namespace App\Domain\Messenger {
    class Message
    {}
}

namespace {
    $messengerMessage = new App\Domain\Messenger\Message;
    var_dump($messengerMessage::class);
}

```

Et il n'est pas rare de voir des espaces de noms à 6, 8 niveaux ou plus. Mais attention : dans votre code ça va vite être moche et difficile à lire. C'est pourquoi vous pouvez préciser en avance à quel espace de noms vous faites référence, en utilisant le mot clé `use` suivi du FQCN complet :

```

<?php

namespace App\Domain\Messenger {
    class Message

```

```

    {}
}

namespace {
    use App\Domain\Messenger\Message;

    $messengerMessage = new Message;
    var_dump($messengerMessage::class);
}

```



Cela fonctionne aussi pour les constantes ou les fonctions qui seraient définies dans d'autres espaces de noms. Ce n'est pas un comportement réservé aux classes.

### 1.2.4 Synthèse

Nous avons vu les deux syntaxes : avec un point-virgule, avec des accolades. En pratique, nous préférons utiliser la syntaxe sans accolade. Une pratique courante est de définir l'espace de nom dans lequel on se trouve au début du fichier, puis de préciser les classes provenant des autres espaces de nom (avec le mot-clé `use`) en dessous. Ensuite, on y met notre code.

Vous pouvez observer cela sur le [code de Symfony sur Github](#) (ne cherchez pas à comprendre le code, mais regardez plutôt où est déclaré le namespace et où les `use` sont faits).

## 1.3 Travail à faire



Déposez dans votre dossier de travail `php-poo` le script `index.php` archivé dans le fichier `Etud.zip` de ce TP.



Dans le cadre du projet MatchMaker, il serait bon de séparer nos classes dans des espaces de noms dédiés, et de les organiser de manière logique. Partons du principe que nos espaces de noms seront préfixés par `App\MatchMaker` pour désigner toutes les classes appartenant à notre application MatchMaker.

Ajoutez autant d'espaces de noms que nécessaire, et placez les classes logiquement sous ces espaces.



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

```

> git add --all
> git commit -m "Spécialisez vos objets avec les espaces de noms"

```

```
, > git push -u origin main
```

## 2 Structurez vos fichiers

Maintenant que nous avons le moyen d'identifier des classes par leur espace de noms, et que cet espace peut représenter une arborescence de répertoires, nous allons structurer et distribuer nos fichiers en suivant cette règle.

On va se fixer une règle supplémentaire : **un fichier pour une classe**. À chaque appel de classe on va charger le fichier associé. Non seulement cela diminue la taille de vos fichiers, mais en prime PHP n'ira charger et analyser que le strict nécessaire au moment de la requête.

Voyons à quoi cette distribution peut ressembler avec l'exemple des messages du chapitre précédent. Le fichier `index.php` :

```
<?php

require_once('Domain/Forum/Message.php');

$forumMessage = new Domain\Forum\Message;

var_dump(get_class($forumMessage));
```

Notre classe `Message` a son propre fichier `Message.php` dans le dossier `Forum`; lui-même à un l'intérieur du dossier `Domain`.

Voici le code de notre fichier `Message.php` :

```
<?php

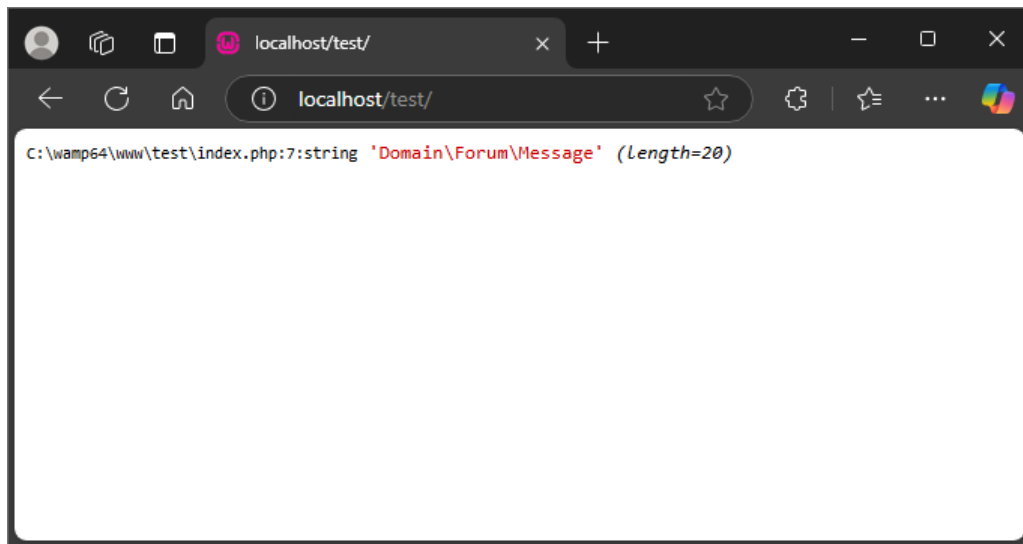
declare(strict_types=1);

namespace Domain\Forum;

class Message
{

}
```





Mais à présent, ajoutons une classe `Utilisateur` dédiée pour définir l'auteur du message. Voyons ce que ça donne, pour le fichier `index.php` :

```
<?php

require_once('Domain/Forum/Message.php');
require_once('Domain/User/User.php');

use Domain\Forum\Message;
use Domain\User\User;

$user = new User;
$user->name = 'Greg';

$forumMessage = new Message($user, 'J\'aime les pates.');
```

```
var_dump($forumMessage);
```

Notre classe `User` a son propre fichier `User.php` dans le dossier `User` ; lui-même rangé à l'intérieur du dossier `Domain` :

```
<?php

declare(strict_types=1);

namespace Domain\User;

class User
{
    public $name;
}
```

Et au passage, nous avons étoffé la classe `Message` :



```

<?php

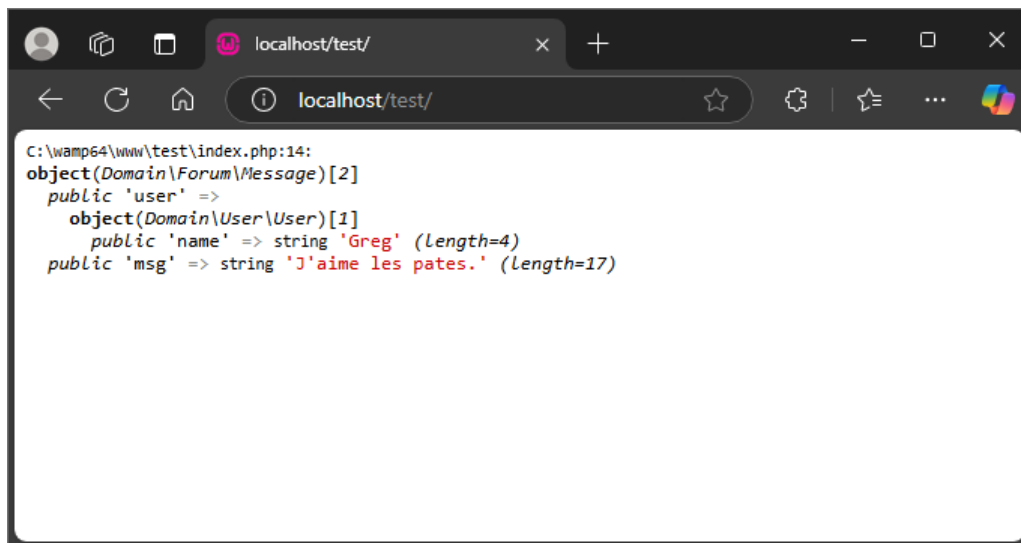
declare(strict_types=1);

namespace Domain\Forum;

class Message
{
    public $user;
    public $msg;

    public function __construct($user, $msg)
    {
        $this->user = $user;
        $this->msg = $msg;
    }
}

```



Le screencast `structurez-vos-fichiers.mkv` explique tout ça en détails.

Nous sommes donc obligés d'utiliser `require_once` pour chaque fichier de chaque classe utilisée... Ça veut dire que pour toutes les classes que j'utilise potentiellement, je suis obligé de faire cet import de fichier. Même si l'utilisation de la classe est dans une condition IF et que la classe n'est pas utilisée systématiquement.

On va donc se retrouver avec un en-tête de fichier très long avec tous les `requires` suivis de tous les `use`. Et comme PHP est un langage interprété ! Il va charger, lire, parser, analyser et interpréter

tous les fichiers demandés à chaque requête que vous faites. Plus votre application va grandir, pire les performances seront.

C'est pourquoi nous allons utiliser une technique grâce à une bibliothèque fournie avec PHP : le chargement automatisé de la bibliothèque standard PHP, [SPL](#).

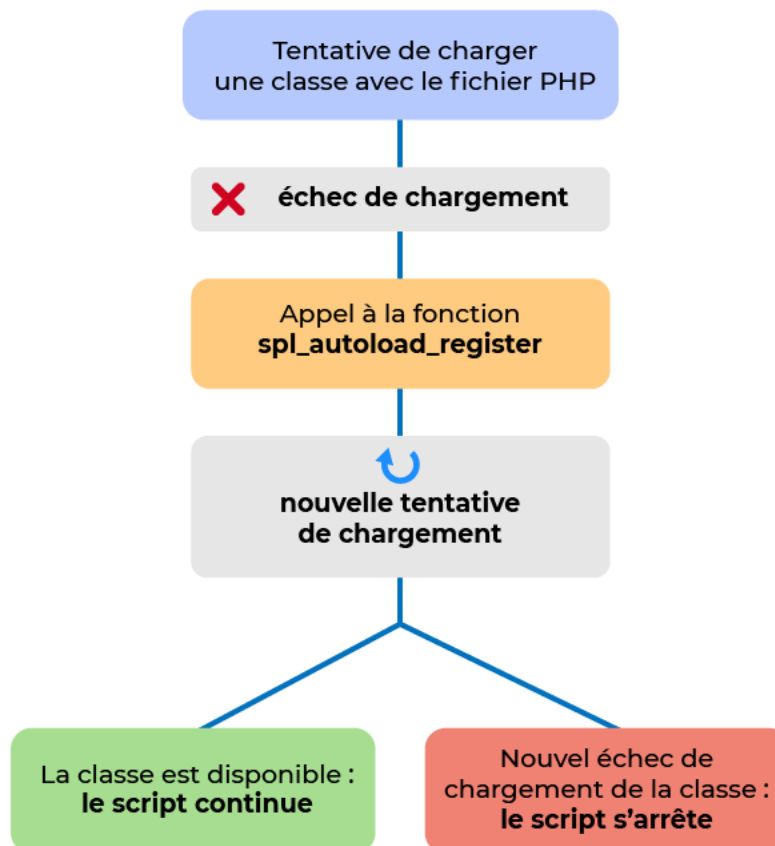
## 2.1 Tirez profit du chargement automatisé

Dans la bibliothèque [SPL](#) se trouve une fonction nommée `spl_autoload_register`.



| Le screencast `composer.mkv` vous donne un premier aperçu de cette mécanique.

Lorsque vous tentez de charger une classe (instanciation, usage de constante, etc) alors que PHP ne la trouve pas, PHP va appeler automatiquement les fonctions enregistrées précédemment à l'aide de `spl_autoload_register`. À la suite de l'exécution des fonctions enregistrées, le script reprend son cours et tente à nouveau de charger la classe. Si le script échoue à nouveau, il s'arrête.



Puisque nous écrivons nos espaces de noms comme des chemins de répertoire, si notre arborescence de fichier correspond, alors on peut **automatiser le chargement des fichiers** :

```

<?php

spl_autoload_register(static function(string $fqcn) {
    // $fqcn contient Domain\Forum\Message
    // remplaçons les \ par des / et ajoutons .php à la fin.
    // on obtient Domain/Forum/Message.php
    $path = str_replace('\\', '/', $fqcn).'.php';

    // puis chargeons le fichier
    require_once($path);
});

use Domain\Forum\Message;
use Domain\User\User;

$user = new User;
$user->name = 'Greg';

$forumMessage = new Message($user, 'J\'aime les pates. ');

var_dump($forumMessage);

```

Ici, nous disons à PHP : "Si tu n'arrives pas à charger une classe, voici la fonction que tu peux exécuter pour tenter de la trouver". La fonction en question débute à la fin de la ligne 3 et fait un `require_once` de la classe à partir de son nom complet. Il se trouve que cette façon de faire est la méthode par défaut dans PHP. On aurait pu se contenter d'écrire :

```

<?php

spl_autoload_register();

use Domain\Forum\Message;
use Domain\User\User;

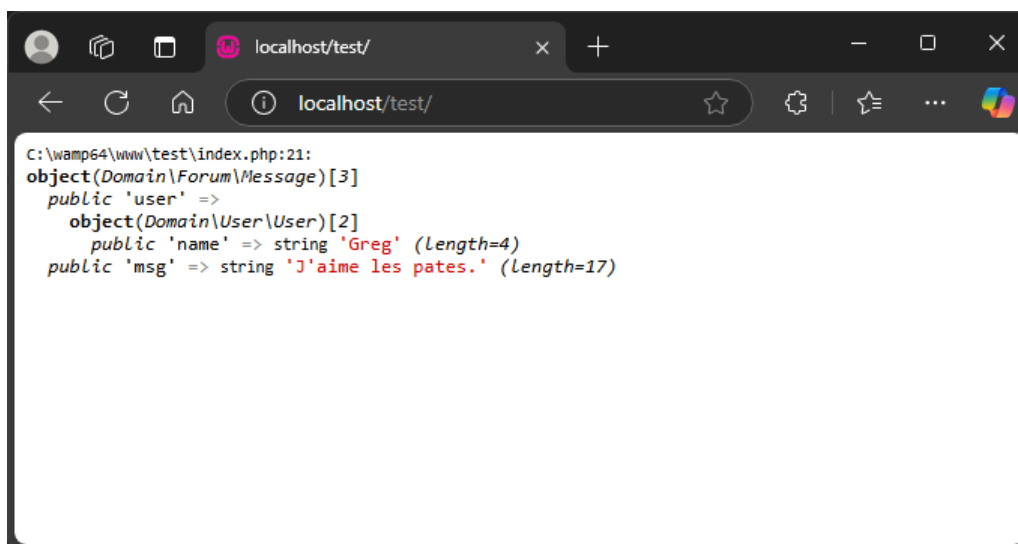
$user = new User;
$user->name = 'Greg';

$forumMessage = new Message($user, 'J\'aime les pates. ');

var_dump($forumMessage);

```



A screenshot of a web browser window with the address bar showing 'localhost/test/'. The page content displays the output of a PHP script, showing a Message object with a user property pointing to a User object. The User object has a name 'Greg' and a message 'J'aime les pates.'.

```
C:\wamp64\www\test\index.php:21:
object(Domain\Forum\Message)[3]
  public 'user' =>
    object(Domain\User\User)[2]
      public 'name' => string 'Greg' (length=4)
      public 'msg' => string 'J'aime les pates.' (length=17)
```

Cette façon de répartir son code en fichiers et répertoires, et d'accorder les espaces de noms, est d'ailleurs très bien détaillée dans la recommandation standard de PHP [PSR-4](#).

## 2.2 Travail à faire



Utilisez la fonction `spl_autoload_register` pour charger automatiquement vos classes, et répartissez-les dans une arborescence correspondant à vos espaces de noms afin de respecter PSR-4.



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

```
> git add --all
> git commit -m "Structurez vos fichiers"
> git push -u origin main
```

## 3 Établissez des contrats avec les interfaces

Une interface se déclare comme une classe, avec le mot clé `interface` au lieu de `class`, et ne peut contenir que des signatures de méthode.

Lorsque vous travaillez en équipe, définir des interfaces sans écrire le code réel permet de se répartir les tâches et de commencer à travailler comme si les classes existaient puisque **vous pouvez préciser une interface comme un type dans vos arguments de méthodes**.

Une interface permet aussi de **spécifier des comportements communs attendus entre différentes classes appartenant à des domaines différents**.

Imaginez une classe `MessagePrinter` dont le rôle est d'afficher un message :

```

1  <?php
2
3
4  class MessagePrinter
5  {
6      public static function printMessage($message)
7      {
8          echo sprintf('%s %s', $message->getContent(), $message->getAuthor()->name);
9      }
10 }

```

Remarquez ligne 6 que l'argument `$message` n'a pas de type. Jusqu'ici, on ne pouvait pas à la fois mettre le type en provenance du domaine `Forum` et celui en provenance du domaine `Messenger`. C'est désormais possible avec PHP 8 en mettant un `|` en séparateur :

```

1  <?php
2
3
4  use Domain\Forum\Message as ForumMessage;
5  use Domain\Messenger\Message as MessengerMessage;
6
7
8  class MessagePrinter
9  {
10     public static function printMessage(ForumMessage|MessengerMessage $message)
11     {
12         echo sprintf('%s %s', $message->getContent(), $message->getAuthor()->name);
13     }
14 }

```

Mais ça ne résout pas vraiment le problème de fond. Si demain une nouvelle classe de message vient s'ajouter, et que je veux pouvoir l'afficher, je vais devoir venir modifier `printMessage` pour y ajouter une classe... Si j'en ai 10, je dois mettre les 10 ? C'est là qu'entre en jeu l'interface.

### 3.1 Syntaxe

Créons donc une interface pour nos messages :

```

<?php

namespace Domain\Display;

use Domain\User\User;

```

```
interface MessageInterface
{
    public function getContent(): string;
    public function getAuthor(): User;
}
```

Toute classe utilisant cette interface sera obligée d'avoir les méthodes `getContent` et `getAuthor` qui renverront une chaîne de caractères et un utilisateur.

On peut maintenant utiliser cette interface en tant que type dans notre méthode `printMessage` :

```
<?php

declare(strict_types=1);

namespace use Domain\Display;

class MessagePrinter
{
    public static function printMessage(MessageInterface $message)
    {
        echo sprintf('%s %s', $message->getContent(), $message->getAuthor()->name);
    }
}
```

Tant que notre code garantit que l'objet passé en argument est issu d'une classe utilisant l'interface, alors ça marchera.

Ajoutons l'interface sur les classes, à présent. Cela s'effectue par le biais du mot clé `implements` suivi du nom de l'interface, en dernier, juste après le nom de la classe. Et si la classe en étend une autre, après le nom de la classe étendue.

```
<?php
declare(strict_types=1);

namespace Forum {
    use Domain\Display\MessageInterface;

    class Message implements MessageInterface
    {
        // ... implémentation des méthodes de l'interface
    }
}

namespace Messenger {
    use Domain\Display\MessageInterface;
```

```

class Message implements MessageInterface
{
    // ... implémentation des méthodes de l'interface
}

```

Contrairement à l'extension de classe, vous pouvez **implémenter plusieurs interfaces en même temps**. Ainsi, la solution précédente aurait pu être la suivante :

```

<?php

namespace Domain\Display {
    use Domain\User\User;

    interface ContentAwareInterface
    {
        public function getContent(): string;
    }

    interface AuthorAwareInterface
    {
        public function getAuthor(): User;
    }
}

namespace Domain\Forum {
    use Domain\Display;

    class Message implements Display\AuthorAwareInterface,
                             Display\ContentAwareInterface {
        // Implémentation des méthodes des interfaces
    }
}

```

Il y a une règle à respecter : une interface par ensemble de méthodes inséparables. On appelle ceci la **ségrégation des interfaces** (⚡ **principes SOLID**). Plus la "surface" de ces interfaces est petite, plus les dépendances demandées dans les autres classes seront ciblées. Cela veut dire que le code n'aura qu'un très faible couplage avec de vraies classes, et que ce sera d'autant plus facile de les refactoriser si nécessaire.



| Ce dernier exemple ne fonctionne plus avec la méthode `printMessage` !

Heureusement qu'une interface peut hériter de plusieurs interfaces avec le mot clé **extends** !

```
<?php

namespace Domain\Display {
    use Domain\User\User;

    interface ContentAwareInterface
    {
        public function getContent(): string;
    }

    interface AuthorAwareInterface
    {
        public function getAuthor(): User;
    }

    interface MessageInterface extends ContentAwareInterface, AuthorAwareInterface
    {
    }
}

namespace Domain\Forum {
    use Domain\Display;

    class Message implements MessageInterface {
        // Implémentation des méthodes des interfaces
    }
}
```

Voilà qui est mieux. Vous pouvez composer vos interfaces pour qu'elles soient le moins couplées possible. Dit autrement, il faut essayer de faire en sorte que **lorsque vous utilisez une interface, vous ayez besoin d'utiliser l'intégralité des méthodes réclamées**. Dans le cas contraire, il serait peut-être bon de séparer les interfaces en de plus petites, plus spécialisées.

### 3.2 Utilisez les interfaces comme type d'argument dans vos méthodes



Le screencast `interfaces.mkv` vous montre comment utiliser les interfaces en tant qu'argument de méthode, et un piège commun dans lequel il vaut mieux ne pas tomber. ?

### 3.3 Travail à faire





Créez des interfaces pour les classes `Player`, `QueuingPlayer` et `Lobby`.  
Puis, remplacez les arguments attendus en utilisant les interfaces.



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

```
> git add --all  
> git commit -m "Établissez des contrats avec les interfaces"  
> git push -u origin main
```

## 4 Évoluez vers la composition



Qu'est-ce que la **composition** ?

Disons que j'ai une classe pour enregistrer un nouvel utilisateur, et qu'à présent je souhaite envoyer un e-mail pour avertir l'utilisateur de la création de son compte.

Au lieu d'ajouter une méthode `envoyerEmail` dans ma classe, ou d'étendre ma classe et d'ajouter cette méthode dans la classe enfant, le choix de la composition nous dicte plutôt de :

1. créer une nouvelle classe dédiée à l'envoi de ce mail,
2. et donner une instance de cette classe à la première.

La mise en place d'une composition ne peut pas s'effectuer simplement au fil de l'eau. Le nombre de classes et la portée de leurs actions doivent être réfléchis en amont. Généralement, ça passe par la création d'interfaces :

- les classes implémentent les interfaces ;
- les interfaces sont développées et ajoutées aux classes ;
- et les classes les réclament soit par nouvelles créations (instanciation), soit par injection en argument de constructeur ou de méthode.



Comment savoir quand choisir l'héritage ou la composition ?

Choisissez **l'héritage** lorsque vous pouvez dire "c'est un". Choisissez la **composition** lorsque vous pouvez dire "possède un/des" ou "est composé de".

La composition est un principe qui donne plus de souplesse à la conception et à l'évolution de votre code. C'est plus naturel de construire des classes en en possédant d'autres qui sont spécialisées, plutôt que d'essayer de trouver des points communs entre elles, et de créer un arbre généalogique. Cette approche permet de s'adapter plus facilement aux modifications futures ! Dans le cas où vous utilisez l'héritage, si vous effectuez un changement relativement important dans une classe située très haut dans l'arbre généalogique, vous impactez tous les enfants. Souvent ça implique de restructurer ces classes-ci, ainsi que celles qui en dépendent... Pour cela, la composition est préférable quand c'est possible.

## 4.1 Composez, structurez et faites dialoguer vos classes



| Le screencast `composition.mkv` vous montre comment vous pourriez structurer votre code. ?

## 4.2 Faites attention à la loi de Demeter

Nous venons de créer des classes, de les structurer et de les faire dialoguer entre elles.

Il arrive que (par fainéantise) l'on rencontre des objets un peu trop ouverts... et qu'on en abuse. Le danger du comportement que je vais exprimer juste après, c'est qu'il est insidieux, car on ne constate les dégâts que très tard ! Quand on doit faire des changements simples et que d'un coup, on se rend compte que l'impact est global sur notre application (j'exagère, mais à peine)...

Le problème survient lorsqu'un objet manipule directement la dépendance d'une de ses dépendances.



| Le screencast `demeter.mkv` illustre ce dernier point.

## 4.3 Travail à faire



Reprenons la classe `QueuingPlayer`. Elle hérite de la classe `Player`. Ce n'est pas idéal, car le fait d'être en attente d'un match ne justifie pas nécessairement d'étendre la classe `Player`. Modifiez la classe `QueuingPlayer` pour que le `Player` soit une propriété au lieu d'un héritage. Si vous devez faire des ajustements à d'autres endroits du code existant, n'hésitez pas.



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

```
> git add --all
> git commit -m "Évoluez vers la composition"
> git push -u origin main
```

## 5 Gérez vos cas d'erreur

### 5.1 Remontez un cas d'erreur avec le mot clé return



Le screencast `erreur1.mkv` explique comment PHP gère les erreurs.

### 5.2 Remontez un cas d'erreur avec le mot clé throw

On gère les cas d'erreurs en jetant une exception qui s'accompagne d'un message et qui peut s'accompagner d'un code d'erreur. Pour cela, il faut utiliser le mot clé `throw`.

Exemple :

```
<?php

/**
 * @var string $text le contenu du message
 * @return bool true en cas de succès
 * @throw Exception on error
 */
function sendEmail(string $text): bool
{
    if (/*on simule que l'envoi du message réussie*/ false)
    {
        // l'exception jetée avec son message et son code d'erreur
        throw new Exception('L\'envoi du mail a échoué',
            '22eb3737-3f43-497e-9912-a737975072ea');
    }

    return true;
}

/**
 * @var string $text le contenu du message
 * @return bool true en cas de succès
 * @throw Exception on error
```

```

*/
function sendNotification(string $text): bool
{
    if (/*on simule que l'envoi de notification échoue*/ true)
    {
        throw new Exception('L\'envoi de la notification a échoué',
            'f3259929-61b2-44d0-a3d6-b855890c0726');
    }

    return true;
}

/**
* @var string $text le contenu du message
* @return bool true en cas de succès
* @throw Exception on error
*/
function sendMessage(string $text): bool
{
    if (10 > strlen($text)) {
        throw new Exception('Le texte est trop court',
            '02bc3998-b9ff-431e-9058-8ab333ff7742');
    }

    sendEmail($text);
    sendNotification($text);

    return true;
}

```

Vous remarquez que nousinstancions des objets `Exception` avec un message et un code d'identification unique, pour qu'un programme puisse les distinguer.

Et dans la fonction `sendMessage` l'appel des méthodes `sendEmail` et `sendNotification` est indépendamment de la vérification de longueur.

### 5.3 Attrapez les exceptions pour mieux les traiter

Pour attraper une exception, il faut utiliser la structure `try { ... } catch (Exception $e) { ... }`.

- Entre les accolades du `try`, on va mettre le code qui potentiellement peut jeter une exception.
- `catch` se comporte comme une fonction, car entre parenthèses on mettra un argument typé `Exception`, qui sera appelé en cas d'exception lancée dans le `try`.

- Dans les accolades du `catch`, le code à exécuter en cas d'interception d'une exception, généralement un moyen de traiter l'erreur reçue.



| Le screencast `erreur2.mkv` montre comment intercepter les exceptions.

On commence à avoir une belle gestion des erreurs, mais on peut faire encore mieux ! `Exception` est une classe, donc... on peut l'étendre.

## 5.4 Les exceptions natives de PHP

Dans PHP, par défaut il n'existe que la classe `Exception`. Mais la librairie SPL, qui est embarquée avec PHP, en propose d'autres.

SPL propose 2 classes qui étendent `Exception` : `LogicException` et `RuntimeException`. Ainsi qu'une dizaine d'autres qui découlent de ces deux-ci, que vous pouvez voir dans la [doc PHP](#).

Lorsque vous voulez expliquer à un développeur qu'il n'a pas correctement utilisé le code, et qu'il doit effectuer un changement dans son code, alors vous devez utiliser une `LogicException`.

Si vous voulez exprimer une erreur suite à une saisie utilisateur, qui ne pourrait pas être détectée autrement que durant l'exécution, et donc qu'un simple message d'explication suffit, alors vous devrez utiliser une `RuntimeException`.

## 5.5 Créez des exceptions personnalisées

En reprenant nos précédents exemples, nous pourrions remplacer les exceptions par des classes dédiées.

Lorsque l'on va attraper les exceptions, on va demander à `catch` de regarder la classe de l'exception obtenue.

Nous avons 3 exceptions à créer :

- une en cas d'erreur d'envoi de mail ;
- une en cas d'erreur d'envoi de notification ;
- et une en cas de texte trop court.

Dans les 2 premiers cas, l'utilisateur ne peut rien y faire. Est-ce qu'il s'agit d'une `LogicException` ? Eh non, c'est un piège. Parce qu'ici le développeur ne peut rien y faire non plus. Nous sommes donc pour chacune dans le domaine des `RuntimeException`.

```
<?php
```

```
class EmailSendingErrorException extends RuntimeException
{
    public $message = 'Impossible d\'envoyer l\'email.';
}
```

```

class NotificationSendingErrorException extends RuntimeException
{
    public $message = 'Impossible d\'envoyer la notification.';
}

class ShortText extends RuntimeException
{
    public $message = 'Le texte fourni est trop court.';
}

```

Puis on remplace le code avec nos exceptions :

```

<?php

class EmailSendingErrorException extends RuntimeException
{
    public $message = 'Impossible d\'envoyer l\'email.';
}

class NotificationSendingErrorException extends RuntimeException
{
    public $message = 'Impossible d\'envoyer la notification.';
}

class ShortText extends RuntimeException
{
    public $message = 'Le texte fourni est trop court.';
}

/**
 * @var string $text le contenu du message
 * @return bool true en cas de succès
 * @throw Exception on error
 */
function sendEmail(string $text): bool
{
    if (/*envoi du message échoue*/ true)
    {
        throw new EmailSendingErrorException();
    }
}

```

```

    return true;
}

/**
 * @var string $text le contenu du message
 * @return bool true en cas de succès
 * @throw Exception on error
 */
function sendNotification(string $text): bool
{
    if (/*envoi de notification échoue*/ true)
    {
        throw new NotificationSendingErrorException();
    }

    return true;
}

/**
 * @var string $text le contenu du message
 * @return bool true en cas de succès
 * @throw Exception on error
 */
function sendMessage(string $text): bool
{
    if (10 > strlen($text)) {
        throw new ShortTextException();
    }

    try {
        sendNotification($text);
    } catch (NotificationSendingErrorException $e) {
        // Envoyez vous une alerte
        // pour vous prévenir que les notifications ne marche pas ;)
    } finally {
        // finally permet d'exécuter du code quoi qu'il arrive :)
        sendEmail($text);
        // si une exception est jetée par sendEmail,
        // Le return n'est jamais exécuté
        return true;
    }
}

try {

```

```

    sendMessage('Hello, ici Greg "pappy" Boyington');
} catch (ShortTextException $e) {
    echo $e->message;
} catch (EmailSendingErrorException $e) {
    echo 'Une erreur est survenue lors de l\'envoi du message,
    nos équipes ont été prévenues, veuillez réessayer plus tard';
} catch (Exception $e) {
    echo 'Une erreur inattendue est survenue, nos équipes ont été prévenues,
    veuillez réessayer plus tard';
}

```

On peut enchaîner les `catch` pour gérer les différents comportements maîtrisés et gérer les cas généraux.

## 5.6 Travail à faire



Décompressez dans votre dossier de travail `php-poo` le contenu du fichier `exception.zip` archivé dans `Etud.zip`.



Pour faciliter la gestion d'erreur, remplacez les usages de la fonction `trigger_error` déjà présente dans le code, par l'usage des exceptions.

Vous créerez un répertoire `Exceptions` dans le répertoire `src`. Et à l'intérieur, vous créerez vos exceptions personnalisées.

Enfin, examinez le code et placez des `try...catch` aux endroits qui vous semblent importants.



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

```

> git add --all
> git commit -m "Gérez vos cas d'erreur"
> git push -u origin main

```