Module 02 - Rappels : Programmation Orientée Objet et principes de design (application en PHP)

> En cas de question, erreur, suggestions, etc. me contacter par mail
> paul.schuhmacher.ext@eduservices.org

Sommaire

Vous retrouverez ces concepts dans tous les langages supportant la Programmation Orienté Objet (*POO*): PHP, Java, C#, C++, Python, Kotlin, Dart, etc.

- Les bases de PHP;
- Classe et objets (`class`, `new`, `__construct()`, `__destruct()`)
- Visibilité (`public`, `private`, `protected`)
- Opérateur de résolution de portée `::`
- `static`
- Héritage (`extends`)
- Contrôle sur l'héritage avec le mot clef `final`
- Classe et méthode abstraite (`abstract`)
- Polymorphisme (*Late binding*)
- Concept général d'interface
- Quelques principes de design

Classes et objets : Un programme composé d'objets

Les programmes designés en POO sont composés d'*objets* qui échangent des messages. L'exécution du programme se traduit par l'échange de messages entre objets. Un message est émis à un objet quand on demande à un objet d'exécuter une de ses méthodes.

```
$foo = new Foo();
$bar = new Bar();
//$foo reçoit le message 'doStuff' dont le contenu est donné par $bar
$foo->doStuff($bar->stuffToDo());
```

> La difficulté d'un système designé avec de la programmation orienté objet est de décomposer le système en objets tout en gardant le système compréhensible et ouvert aux changements (flexibilité).

> En savoir plus

Classes et objets

La classe donne une définition (signature) des propriétés et des méthodes ainsi qu'une implémentation (ce que font les méthodes, le *corps* des méthodes). Une classe définit également un espace de nom.

```php
class Foo{

    //Une propriété (une variable de classe)
    string $proprerty;
    function __construct($property){
        //La pseudo variable $this (mise à disposition par PHP) est disponible lorsqu'une méthode
        //est appelée depuis un contexte objet. $this a pour valeur l'objet appelant.
        $this->property = $property;
    }
    //Une méthode (une fonction de classe)
    function doStuff(array $stuffToDo){
        //instructions
    }
}
```

Classes et objets : Instancier un objet `new` et accéder aux propriétés/méthodes

Un objet est une instance de classe, on instancie un objet avec l'opérateur `new`, qui fait appel au constructeur `__construct()` pour initialiser l'objet.

```
//Création d'un objet
$foo = new Foo('foo');
//Accéder à une propriété (l'encapsulation est brisée, les détails internes de l'objet sont exposés à sa surface et accessibles au monde entier)
$foo->property;
//Accéder à une méthode (ie envoyer un message à l'objet)
$foo->doStuff(array(1,2,3))
```

Dans un langage POO, envoyer des messages (appeler des méthodes) est la seule manière à notre disposition pour faire agir un objet (exécuter du code).

Les messages sont (et *devraient!*) être la seule manière de changer l'état interne d'un objet. Ainsi, l'état interne d'un objet est dit *encapsulé*, ses rouages internes (propriétés, méthodes privées, etc.) sont invisibles depuis l'extérieur.

Classes et objets : Signature d'une méthode

Une méthode possède une signature c'est-à-dire :

- Une visibilité;
- Un type de retour (`void` indique que la méthode ne retourne rien);
- Un ou plusieurs arguments ayant chacun un type (`void` indique que la méthode ne prend rien en argument);

```
//La signature de la méthode `doStuff` de la classe Foo
public doStuff(array $stuffToDo): void
```

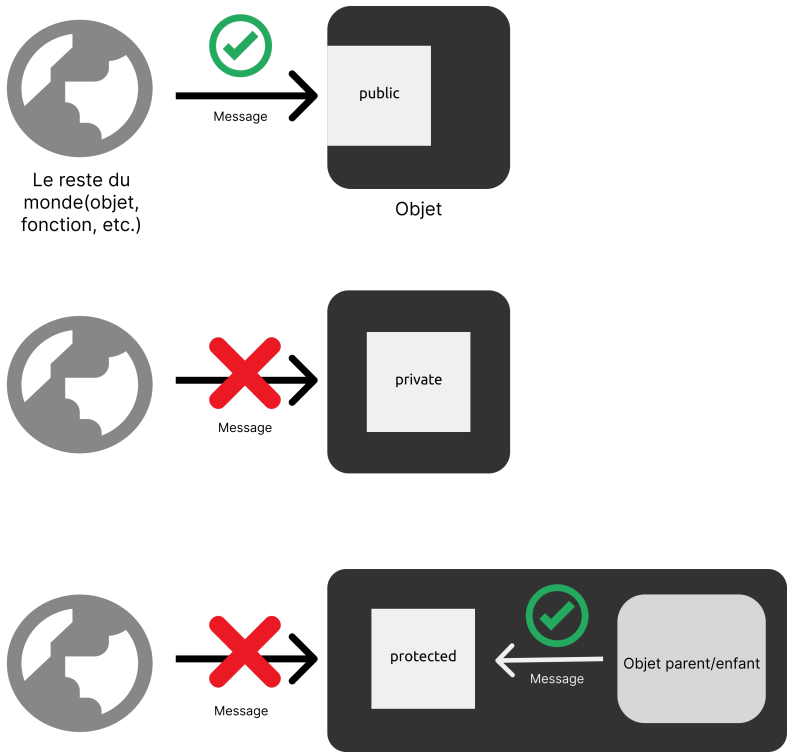
Classes et objets : Visibilité

La visibilité définit l'exposition des attributs et des méthodes d'un objet aux autres objets dans le programme. Il existe 3 modificateurs de visibilité:

- `public` : c'est la visibilité par défaut en PHP. Un attribut/une méthode `public` est accessible depuis l'extérieur de la classe, par tous les objets du programme.
- `private` : un attribut/une méthode `private` n'est pas accessible depuis l'extérieur de la classe. Aucun objet dans le programme ne peut y accéder, à part l'objet qui les porte. On ne peut pas redéfinir une propriété/méthode privée dans une classe enfant.
- `protected` : un attribut/une méthode `protected` n'est pas accessible depuis l'extérieur de la classe sauf par les classes parent ou enfant.

> En savoir plus

Classes et objets : Visibilité



Opérateur de résolution de portée

L'opérateur de résolution de portée `::` donne un moyen d'accéder aux membres `static`, `const` (constantes de classe) ainsi qu'aux méthodes surchargées.

- Depuis l'intérieur de la classe: `self::`, `parent::`, `static::`
- Depuis l'extérieur de la classe: `NomDeLaClasse::`

> En savoir plus

Opérateur de résolution de portée

```php
class Foo
{
    public function __construct()
    {
        echo 'Call Foo constructor' . PHP_EOL;
    }
}

class Bar extends Foo
{
    const BAZ = 'baz';
    public function __construct()
    {
        //Appel au constructeur parent
        parent::__construct();
        echo 'Call Bar constructor' . PHP_EOL;
    }
}
$bar = new Bar();
echo Bar::BAZ . PHP_EOL;
```

Attributs et méthodes `static`

`static` est un mot-clef historiquement utilisé par le C++ et utilisé par les autres langages. Dans ce contexte il signifie "de classe".

Une propriété ou une méthode `static` peut être accédée sans avoir besoin d'instancier la classe (*statique,* car pas besoin pour y accéder d'utiliser l'opérateur `new` et une allocation mémoire, c'est un accès *à froid*)

```php
class Foo
{
    private static string $bar = 'bar';
    public static function bar()
    {
        return static::$bar;
    }
}
//Je peux appeler la méthode bar de la classe Foo sans instancier d'objet Foo
echo Foo::bar();
```

> En savoir plus. Remarque: on parle d'allocation statique car la mémoire est réservée

L'héritage

Une classe peut étendre une autre classe avec l'expression `extends` afin d'hériter de ses propriétés et de ses méthodes. Une classe *enfant* peut accéder et redéfinir les attributs/méthodes de sa classe parente (sauf si `private`). L'héritage est un **mécanisme de partage de code**.

```
class Foo{
    private $baz
    public function baz(){}
}
//Bar étend Foo, Bar est une classe enfant de Foo
class Bar extends Foo{
    //Bar dispose de la propriété $baz et de la méthode baz()
}
```

L'héritage est vu souvent comme une "*is-a*" relation entre deux classes, mais c'est surtout une "*behaves like*" relation. Car la classe enfant hérite de l'**implémentation** (comportement) de sa classe parent.

> L'héritage crée un *couplage fort* entre deux classes, tout en exposant les détails internes de la classe parent à ses classes enfant.À utiliser avec précaution. En savoir plus sur l'héritage

Héritage : contrôle sur l'héritage avec `final`

On peut contrôler l'héritage avec le mot clef `final`. Une classe `final` est une classe qui ne peut pas être étendue.

```
final class Foo{}
//Erreur, Foo ne peut pas être étendue
class Bar extends Foo{}
```

Héritage : contrôle sur l'héritage avec `final`

Une méthode `final` est une méthode qui ne peut pas être redéfinie dans une classe enfant.

```
class Foo(){
    public final function foo(){ echo 'Foo call foo';}
    public function bar(){ echo 'Foo call bar';}
}
class Baz extends Foo{
    //Erreur, impossible
    public function foo(){ echo 'Baz call foo';}
    //Valide
    public function bar(){ echo 'Baz call bar';}
}
```

> En savoir plus

Méthodes et classes abstraites avec `abstract`

Les *méthodes abstraites* définissent simplement la signature de la méthode, non leur implémentation, comme une méthode définie dans une interface. Cette signature doit être respectée par l'implémentation.

Toute classe définissant une méthode abstraite est abstraite par définition. Une *classe abstraite* ne peut pas être instanciée, car son implémentation est incomplète !

Méthodes et classes abstraites avec `abstract`

À la différence des `interface`, une classe abstraite peut être partiellement implémentée et disposer de propriétés comme n'importe quelle classe.

```php
abstract class Foo{
    protected int $foo;
    abstract public function foo():int;
    public function bar(){echo 'bar' . PHP_EOL;}
}
class Bar extends Foo{
    //Bar définit l'implémentation de la méthode abstraite foo, en respectant sa signature
    public function foo():int {
        echo 'Implémentation de foo dans la classe Bar' . PHP_EOL;
        return 10;
    }
}
$bar = new Bar();
echo $bar->foo() . PHP_EOL;
$bar->bar();
```

> En savoir plus

Les interfaces: `interface` et `implements`

Une interface PHP permet de définir des méthodes (*signatures*) qu'une classe *doit* implémenter (`implements`) sans avoir à définir *comment* ces méthodes fonctionnent (l'implémentation).

```php
interface Foo{
    public function foo(string $value): string;
}
interface Bar{
    public function bar(string $value): string;
}
//Une classe peut implémenter plusieurs interfaces
class Baz implements Foo, Bar{
    //Implémentation à fournir
    public function foo(string $value): string{
    //instructions..
    }
    public function bar(string $value): string{
    //instructions..
    }
}
```

> Une `interface` est un cas particulier de classe abstraite *sans propriétés ni implémentations*.
>
> En savoir plus

Le polymorphisme

Le *polymorphisme* est la capacité, pour un même message, de correspondre à plusieurs formes de traitements selon l'objet auquel ce message est adressé.

```php
interface A{
    function a();
}
class Foo implements A{
    function a(){echo 'Foo';}
}
class Bar implements A{
    function a(){echo 'Bar';}
}
//le message a est envoyé à $foo, il affiche 'Foo' (traitement de Foo)
$foo = new Foo()->a();
//le même message a est envoyé à bar, il affiche 'Bar' (traitement de Bar)
$bar = new Bar()->a();
```

Interface et type

L'interface d'un objet (on ne parle pas ici du construct PHP `interface`) désigne **l'ensemble des signatures publiques des méthodes d'un objet**. N'importe quel message qui correspond à une signature de l'interface peut être envoyé à cet objet. L'interface d'un objet est synonyme de **type** de l'objet.

Dans un système OO *où l'encapsulation est bien respectée*, l'interface est *fondamentale,* car les objets sont connus (comment les utiliser) seulement *via* leurs interfaces (ce qu'ils exposent à leur surface).

Interfaces et types

- Un objet peut avoir plusieurs *types* (implémenter plusieurs interfaces);
- Un type est *sous-type* d'un autre si son interface contient l'interface du *super-type*;
- Deux objets partageant la même interface peuvent **agir de manière complètement différente** (avoir des implémentations différentes);

```php
interface A{
    function a();
}
interface B{
    function b();
}
//Foo est de 'type' A, car il possède l'interface de A
class Foo implements A{function a(){}}
//Bar est de type A ET de type B, car il possède les deux interfaces.
//Je peux lui envoyer le message `b` (i.e appeler sa méthode b) et le message `a`.
//On peut aussi dire que Bar est un sous-type de A (super-type), car son interface contient l'interface de A.
class Bar implements A, B{function a(){} function b(){} function bar(){}}
//Baz déclare une propriété de type A, une implémentation de A lui est fournie par injection de dépendance.
class Baz {private A $a; function construct(A $a){$this->a = $a;} function a(){$this->a->a();}}
```

> Un diagramme de classe UML permet de bien visualiser les interfaces des objets

Principes de design, les design patterns *de premier niveau*

Objectifs

L'architecture logicielle est l'ensemble des techniques et concepts qui vous aident à *organiser* votre système pour qu'il soit:

- *Simple à comprendre*: je peux raisonner sur son fonctionnement sans avoir à penser à *trop* de choses à la fois;
- *Simple à modifier*: je peux changer, ajouter, retirer une partie de mon système sans que cela n'ait trop d'impact dessus;

> Fabriquer un système *simple* est une chose *difficile* !

Voyons quelques principes de design pour la POO.

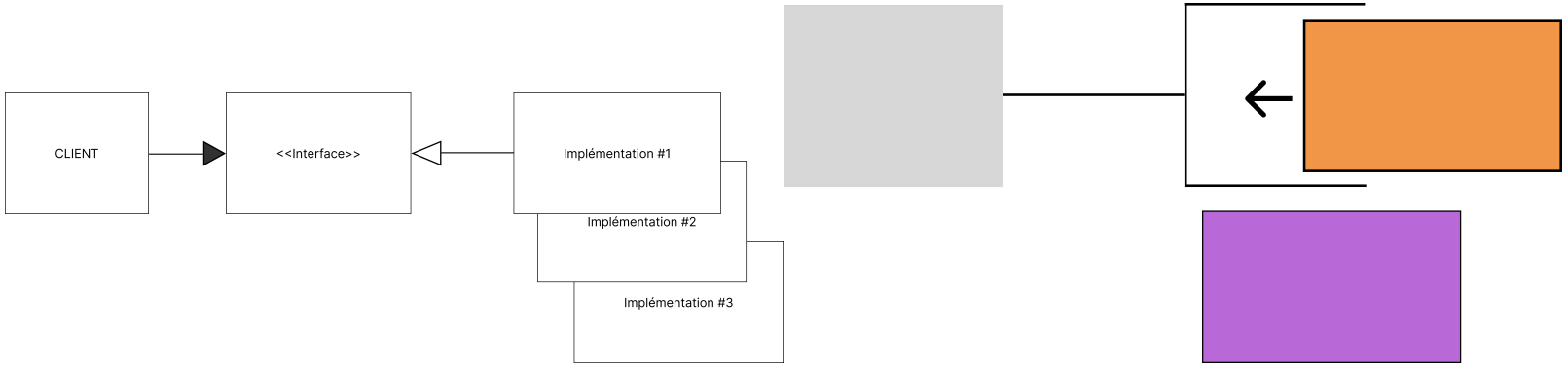
Isoler ce qui change de ce qui ne change pas (trop)

Si l'un des aspects de votre code est susceptible de changer, vous savez que vous êtes face à un comportement qu'il faut isoler du reste de votre système. Vous pourrez ainsi les modifier plus tard sans affecter ce qui ne varie pas.

Programmez vers une interface, pas une implémentation

Si vous programmez vers une interface (c'est-à-dire que votre code client consomme des interfaces et non des implémentations) vous pouvez envoyer le même message aux objets que le client manipule (appeler les mêmes méthodes) tout en obtenant un comportement différent. Vous pouvez ainsi simplement *changer d'implémentation* en fonction de vos besoins.

Ce principe dit donc *utilisez le polymorphisme*.



Préférez la composition à l'héritage (quand c'est possible)

Passez de la relation *est-un* et surtout *se comporte comme* (héritage) à la relation *a-un* (composition) entre deux classes.

- Beaucoup plus flexible, vous pouvez modifier les comportements au *run-time* (à l'éxecution), alors qu'avec l'héritage vous devez le faire *at compile-time* (avant exécution), d'où cette idée de couplage fort entre les deux classes. Un changement dans la classe parent amène souvent des changements dans la classe enfant.
- L'héritage vous fait hériter de l'interface, *mais aussi de son implémentation !* On parle alors de *White box visibilité* (détails internes partagés, encapsulation brisée)
- La *composition* (un objet garde une référence vers d'autres objets dans une propriété) offre une *Black box visibility* (aucun détail interne partagé). Un objet composé d'un autre objet n'a besoin de connaître que son interface pour s'en servir (encapsulation préservée).

Exercices

Voir fiche d'exercices fournie.

Ressources supplémentaires

> Un lien vers la documentation officielle pour chaque concept exposé est présent sur chaque page

- Manuel PHP, la doc officielle de PHP traduite en français, très complète (explications, exemples)
- Développons en JAVA, Jean Michel DOUDOUX, Jean-Michel Doudoux est une référence dans la communauté Java francophone, son site présente une documentation très complète et très bonne sur Java (maintenue depuis 2001!), mais également sur tous les concepts de la POO.

Aller plus loin

> Lectures et visionnages recommandées

- Solid Principles, *Uncle Bob* [en anglais], Robert C. Martin, Uncle Bob, discute des principes SOLID notamment de l'architecture plugin et du polymorphisme;
- Design Patterns: Elements of Reusable Object-Oriented Software, de Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, aka "the Gang of Four (GoF)", publié chez Addison-Wesley, 1994. La source. Trouvable facilement en PDF sur le web;
- Chapitre *How Design Patterns Solve Design Problem* de *Design Patterns: Elements of Reusable Object-Oriented Softwares*, **pages 24 à 36** [en anglais]. Cette section est très bien faite et décrit ce qu'est une interface (sous-section *Specifying Object Interfaces*) ainsi que les fondamentaux de la programmation orientée objet. **Lecture recommandée**;

Aller plus loin

> Lectures et visionnages recommandées

- The computer revolution hasnt happened yet, conférence d'Alan Kay at OOPSLA 1997, toujours très transversal, sur les origines de la programmation orientée objet et ses ambitions initiales (produire des systèmes à grande échelle qui durent dans le temps, comme le réseau Internet);
- Alan Kay Object Oriented Programming "OOP" Talk, conférence d'Alan Kay sur la programmation orientée objet et ses intentions;