Sujet 3 : Implémentation d'un objet simple et encapsulation des données

Avant de commencer

Dans le répertoire `POO` créez un sous-répertoire `TP3`. Vous y stockerez l'ensemble des solutions développées tout au long de ce TP. Votre enseignant devra y retrouver la trace de tout le travail effectué, ne remplacez donc pas vos solutions par d'autres, privilégiez, en cas de nécessité, une mise en commentaire. Pour une question de lisibilité, veillez à reporter les numéros des questions dans votre code. De la même façon les noms des auteurs devront apparaître en haut de chaque script. Les réponses aux questions *d'ordre théorique* (qui ne sont pas du code) seront enregistrées dans un fichier TP3.txt.

Création du dépôt Git

Votre travail sera impérativement déposé sur un dépôt Git. Si vous avez opté pour un dépôt commun à tous les TPs, n'oubliez pas de le compléter systématiquement en créant un répertoire par sujet de TP. Dans le cas contraire initialisez un nouveau dépôt en suivant les directives d'initialisation de dépôt décrites ci-dessous.

Consignes

Vous veillerez à respecter scrupuleusement les consignes suivantes :

- Vous effectuerez un commit après chaque exercice.
- A la fin de chaque séance, vous effectuerez un commit. Si ce commit contient du code incomplet ou ne fonctionnant pas, mentionnez-le dans le message de commit. Vous pousserez ensuite votre travail vers le dépôt distant.
- Ce dépôt sera utilisé par votre enseignant(e) de TP pour évaluer votre travail. Assurez-vous donc régulièrement que tous les fichiers que vous souhaitez lui rendre sont bien présents dans le dépôt.
- Le dépôt lui-même sera évalué : soignez l'écriture de vos messages.

Initialisation du dépôt

- Ouvrez un terminal et utilisez la commande *cd* pour vous placer dans le répertoire $TP3$
- Initialisez le dépôt *Git* du projet.
- Si nécessaire, éditez le fichier *.gitignore*
- Effectuez un premier commit
- Connectez-vous à l'application *Gitlab* et créez un dépôt nommé $POO - TP3$
- Assignez le rôle *Reporter* à votre enseignant(e) de TP.
- Poussez le dépôt local vers le dépôt distant.

Liens utiles

- Aide-mémoire Git : *https://iut-info.univ-reims.fr/users/nourrit/git_aide_memoire.pdf*
- Gitlab : *https://iut-info.univ-reims.fr/gitlab*

Les bons usages

D'après les recommandations PSR-2, un script PHP débute par la balise <?*php* mais ne se termine par aucune balise. Vous éviterez de fermer vos scripts avec la balise ? >.

Afin que le typage de vos fonctions et de vos méthodes soit vérifié de manière stricte nous procéderons dorénavant à une typage strict. Vous ajouterez pour cela la directive suivante en tant que première instruction de vos scripts :

```
declare(strict_types = 1);
```

Introduction

L'objectif de ce sujet est d'implémenter des classes simples en préservant le principe d'encapsulation des données. Cet aspect sera traité au travers d'objets présentant des dépendances entre leurs attributs d'instance ou ayant un attribut étant lui-même un objet modifiable.

Exercice 1 : La documentation de la classe Person

Dans le répertoire *TP3*, dupliquez la classe `Person` définie dans le TP2. L'objectif de ce premier exercice sera d'apprendre à générer une documentation propre et lisible pour chacune des classes que vous allez développer. Cette étape est cruciale car elle permet à un utilisateur de comprendre le fonctionnement d'une classe de manière intuitive.

Vous générerez votre documentation avec *phpdoc*. Pour ceci vous ferez précéder chaque définition de méthode de la classe d'un commentaire dédié à la génération automatique de cette documentation. Vous penserez aussi à documenter la classe. Ce type de commentaire est délimité par `/**` et `*/`. Ce commentaire contient une brève description de la méthode, sans pour autant rentrer dans les détails de l'implémentation. On y liste ensuite tous les paramètres de la méthode. Chacun des paramètres est précédé par le mot-clé `@param`, son nom ainsi que de sa description. Pour finir, le retour de la méthode est notifié par le mot-clé `@return` suivi de la description de ce dernier.

Voici un exemple de documentation que l'on pourrait proposer pour le constructeur et l'accesseur de la classe `Person` :

```
/**
 * Constructeur de la classe Person. Ce constructeur permet d'affecter un nom, un prénom
 * et un age à une personne. Lorsque ces caractéritiques ne sont pas renseignées lors de
 * l'appel du contructeur, la personne aura pour nom "Doe", prénom "John" et age 0.
 *
 * @param string $lastName (optional) Nom de la personne
 * @param string $firstName (optional) Prénom de la personne
 * @param int $age (optional) Age de la personne
 */
public function __construct(string $lastName = "Doe", string $firstName = "John",
                                                                        int $age = 0)
{
    $this->lastName = $lastName;
    $this->firstName = $firstName;
    $this->age = $age;
}
/**
 * Accesseur au nom de la personne. Retourne la valeur du nom sous forme de chaîne de
 * caractères.
 *
 * @return string Nom de la personne
 */
public function getLastName() : string
{
    return $this->lastName;
}
```

Question 1.
En vous basant sur les deux exemples ci-dessus, proposez une documentation complète de la classe `Person`.

Maintenant que votre code est clairement commenté il est temps de générer une documentation associée à votre classe. Cette génération se passera à partir du terminal. Vous vous placerez pour ceci dans le répertoire contenant les fichiers à documenter. Voici le ligne de commande permettant d'obtenir un rendu similaire à celui du site PHP :

```
phpdoc -t Documentation --visibility public -f Person.php
```

où `Documentation` sera le répertoire contenant la documentation de la classe `Person`.

Exercice 2 : La classe Ticket de pesée

Dans le répertoire *TP3*, créez un script nommé `WeighingTicket.php`. Il contiendra la définition de la classe `WeighingTicket` permettant de modéliser le concept de ticket de pesée. Éditez ce fichier et complétez-le au fur et à mesure. La documentation de la classe est OBLIGATOIRE et devra systématiquement être générée avec *phpdoc*.

En parallèle créez un script `TestWeighingTicket.php`, n'oubliez pas de commencer votre script par l'instruction : `require_once "WeighingTicket.php";`
Vous compléterez ce script et l'exécuterez à chaque ajout d'une nouvelle méthode d'instance dans la classe `WeighingTicket`.

Les attributs d'instance de la classe

Un ticket de pesée `WeighingTicket` est caractérisé par le nom de l'article concerné par le ticket (`articleName`), son prix au kilogramme (`pricePerKilogram`), son poids en grammes (`weight`) ainsi que son prix réel après pesée (`price`). Ci-dessous vous trouverez le diagramme de classe correspondant :

WeighingTicket
- articleName : string - pricePerKilogram : float - weight : int -/ price : float

Question 2.
Dans le répertoire *TP3*, éditez le script nommé `WeighingTicket.php`. Définissez-y les attributs d'instance de cette classe.

Le constructeur

On souhaite pouvoir créer un objet de type `WeighingTicket` de quatre façons différentes :

```
$inconnu   = new WeighingTicket;
```

```
$carotte   = new WeighingTicket("Carotte");
$concombre = new WeighingTicket("Concombre",0.99);
$pommeDT   = new WeighingTicket("Pomme de Terre",1.5,3000);
```

Le premier appel au constructeur initialisera les attributs d'instance avec pour valeurs respectives : "Unknown", 0, 0 et 0. Le deuxième appel au constructeur utilisera les paramètres pour initialiser les attributs `articleName` et `pricePerKilogram` et initialisera les deux autres à 0. Le troisième appel au constructeur utilisera les paramètres pour initialiser les attributs `articleName`, `pricePerKilogram` et `weight`. La valeur de l'attribut `price` y sera calculée.

Question 3.
Donnez la définition (code) de ce constructeur.

Est-il judicieux de proposer un constructeur prenant en paramètre les valeurs des quatre attributs d'instance ? Si oui complétez la définition précédente, sinon expliquez pourquoi (vous pourrez expliquer pour ceci en quoi réside la spécificité de l'attribut d'instance `price`).

Instanciez les trois objets de l'exemple dans le script `TestWeighingTicket.php`. Exécutez-le.

Les accesseurs et modificateurs

Dans la classe `WeighingTicket`, on souhaite pouvoir accéder et modifier le contenu des attributs d'instance d'une instance de cette classe.

Question 4.
Donnez la définition des quatre accesseurs. Appelez l'ensemble des accesseurs dans le script `TestWeighingTicket.php`. Exécutez-le.

Question 5.
Donnez la définition des modificateurs des attributs d'instance `articleName`, `pricePerKilogram` et `weight`. Faites attention de préserver les dépendances existant entre les différents attributs d'instance. Est-il judicieux de proposer un modificateur de l'attribut d'instance `price` ? Si oui donnez en la définition, sinon expliquez pourquoi.

Appelez l'ensemble des modificateurs dans le script `TestWeighingTicket.php`. Exécutez-le.

Les méthodes (autres)

On souhaite pouvoir afficher une instance de l'objet `WeighingTicket`. Le rendu désiré pour l'instance `inconnu` est :

```
********************
*  Article      : Unknown
*  Prix au kilo : 0
*  Poids (gr)   : 0
*
*  Prix         : 0
********************
```

On souhaite pour ceci pouvoir procéder de la manière suivante :

```
echo "$inconnu\n";
```

Quelle méthode d'instance faut-il alors définir dans la classe `WeighingTicket` ? Définissez-la sans vous inspirer des méthodes précédemment implémentées.

Complétez le script `TestWeighingTicket.php` et exécutez-le.

Question 6.
On considère la méthode d'instance de la classe `WeighingTicket` suivante :

```
public function weighing () : void
{
    $pds = rand(0,5000);
    $this->setWeight($pds);
}
```

Que permet de réaliser la méthode `weighing` ? Commentez la méthode `weighing`. Complétez le script `TestWeighingTicket.php` et exécutez-le.

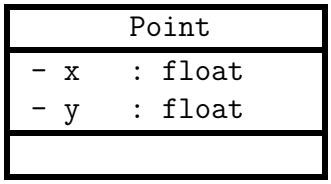
Exercice 3 : Les classes Point et Segment

Dans cet exercice on souhaite définir la notion de segment de droite dans un espace réel à deux dimensions. Un segment est défini par deux points qui se trouvent à ses extrémités. Dans cet exercice vous serez amené à implémenter les classes `Point` et `Segment`.

Dans répertoire *TP3*, créez les scripts `Point.php`, `Segment.php` ainsi que le script de tests `TestPointSegment.php`. La documentation des classes est OBLIGATOIRE et devra systématiquement être générée avec *phpdoc*.

La classe Point

Un `Point` est caractérisé par sa coordonnée en `x` et sa coordonnée en `y`. Ci-dessous vous trouverez le diagramme de classe correspondant :



Question 7.
Définissez les attributs d'instance dans la classe `Point`.

Question 8.
On souhaite pouvoir créer un objet de type `Point` de deux façons différentes :

```
$origine     = new Point;
$destination = new Point( 0.5 , 6.0 );
```

Le premier appel du constructeur initialisera les attributs d'instance à 0. Écrivez la définition de ce constructeur. Complétez le script `TestPointSegment.php` et exécutez-le.

Quelle est la troisième façon d'appeler ce constructeur. Quel est son défaut ? Justifiez votre réponse.

Question 9.
On souhaite disposer des accesseurs sur les deux attributs d'instance de la classe `Point`. Donnez la définition de ces deux accesseurs. Complétez le script `TestPointSegment.php` et exécutez-le.

Question 10.
On souhaite pouvoir afficher une instance de la classe `Point`. Le rendu désiré pour l'instance `origine` est :

```
( 0 , 0 )
```

On souhaite pour ceci pouvoir procéder de la manière suivante :

```
echo "$origine\n";
```

Définissez la méthode d'instance adéquate dans la classe `Point` sans vous inspirer d'une autre implémentation. Complétez le script `TestPointSegment.php` et exécutez-le.

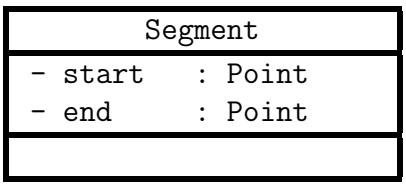
Question 11.
On souhaite pouvoir déplacer un `Point`. Pour ceci on souhaite disposer de la méthode d'instance `translation` dont le prototype est :

```
public function translation(float $dx, float $dy) : void
```

En faisant appel à cette méthode, l'attribut `x` de l'instance sera augmenté de `dx` et l'attribut `y` de `dy`. Donnez la définition de la méthode d'instance `translation`. Complétez le script `TestPointSegment.php` et exécutez-le.

La classe Segment

Un segment est défini par les deux points qui se trouvent à ses extrémités. Un objet de type `Segment` est défini par ses points de début `start` et de fin `end`. Ci-dessous vous trouverez le diagramme de classe correspondant :



Question 12.
Définissez les attributs d'instance dans la classe `Segment`.

Question 13.
On souhaite pouvoir créer un objet de type `Segment` de la manière suivante :

```
$segment1 = new Segment( $origine , $destination);
```

Le constructeur avec initialisation prend en paramètre deux instances de la classe `Point`. Elles lui permettent d'initialiser les deux attributs d'instance tout en veillant à ce que les données soient correctement encapsulées, c'est-à-dire qu'on veillera à ce qu'il n'y ait pas de *partage de la mémoire*. Donnez la définition de ce constructeur. Complétez le script `TestPointSegment.php` et exécutez-le.

Question 14.
On souhaite pouvoir afficher une instance de la classe `Segment`. Le rendu désiré pour l'instance `segment1` est :

```
[ ( 0 , 0 ) - ( 0.5 , 6 ) ]
```

Définissez la méthode d'instance `__toString` dans la classe `Segment`. Dans cette méthode vous veillerez à réutiliser la méthode `__toString` présente dans la classe `Point`. Complétez le script `TestPointSegment.php` et exécutez-le.

Question 15.
On souhaite disposer de l'accesseur sur l'attribut d'instance `start` de la classe `Segment`. Voici une proposition de définition de cet accesseur :

```
public function getStart () : Point
{
   return $this->start ;
}
```

Est-ce que cet accesseur permet de modifier l'attribut `start` de l'extérieur de l'objet ? Justifiez votre réponse.

Afin de tester que votre accesseur respecte le principe d'encapsulation des données, ajoutez les lignes suivantes dans le script `TestPointSegment.php` et exécutez-le :

```
echo "$segment1\n";
$pointDebut = $segment1->getStart();
echo "$pointDebut\n";
$pointDebut->translation(2.,3.);
echo "$pointDebut\n";
echo "$segment1\n";
```

L'instance `segment1` a-t-elle été modifiée après modification de l'instance `pointDebut` ? Si c'est le cas, la définition de cet accesseur est incorrecte. Rectifiez la définition de cet accesseur si nécessaire. Exécutez à nouveau le script `TestPointSegment.php`.

Question 16.
On souhaite disposer du modificateur de l'attribut d'instance `start` de la classe `Segment`. On veillera à ce qu'une instance ne puisse pas être modifiée de l'extérieur (respect du principe d'encapsulation des données). Donnez la définition de ce modificateur. Complétez le script `TestPointSegment.php` et exécutez-le.

Question 17.
On souhaite pouvoir déplacer un `Segment`, ce qui revient à déplacer les deux points qui le définissent. Pour ceci on souhaite disposer de la méthode d'instance `translation` dans la classe `Segment` dont le prototype est le même que celui de la méthode `translation` définie dans la classe `Point`. Donnez la définition de la méthode d'instance `translation`. Complétez le script `TestPointSegment.php` et exécutez-le.

Question 18.
On souhaite disposer d'une méthode d'instance `getLength` dans la classe `Segment` dont le prototype est : `public function getLength () : float`. Cette méthode retourne la distance entre les deux points qui définissent un objet de type `Segment`[1], c'est à dire la longueur du segment. Donnez la définition de la méthode d'instance `getLength`. Vous pourrez pour ceci faire appel à la fonction `sqrt`. Complétez le script `TestPointSegment.php` et exécutez-le.

Question 19.
Dans la classe `Segment` on souhaite disposer d'une méthode d'instance `isLongerThan` dont le prototype est :

```
public function isLongerThan( Segment $segment2 ) : bool
```

Cette méthode retourne *true* si l'objet qui appelle la méthode est plus long que celui passé en paramètre. Donnez la définition de la méthode d'instance `isLongerThan`. Complétez le script `TestPointSegment.php` et exécutez-le.

Question 20.
Sachant que deux segments sont égaux si les points les définissant sont égaux et que l'ordre de ces points n'importe pas, définissez la méthode d'instance `isEqual` dans la classe `Segment`. Cette méthode retournera *true* si deux segments sont égaux, $false$ sinon. Complétez le script `TestPointSegment.php` et exécutez-le.

Question 21.
Afin de créer une copie d'un objet en PHP il est possible de contourner l'instanciation d'un nouvel objet par le biais du constructeur. On peut pour ceci faire appel au mot-clé `clone`, qui lui fait appel à la méthode magique `__clone()` de l'objet, si elle a été définie. La méthode `__clone()` d'un objet peut être appelée *magiquement* :

```
$copieOrigine = clone $origine ;
```

Lorsqu'un objet est cloné, PHP effectue automatiquement une copie superficielle de tous les attributs d'instance de l'objet. Tous les attributs d'instance dont le type est scalaire ou *array* seront dupliqués. Par contre les attributs qui sont eux-même des instances ne vont pas être

[1] Si un segment est défini par les points $P_1(x_1, y_1)$ et $P_2(x_2, y_2)$, sa longueur est $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

dubliqués : des références à d'autres variables demeureront des références, c'est à dire que ces attributs seront partagés entre l'objet d'origine et sa copie. Afin d'éviter ceci et si on le souhaite, il est possible de forcer la copie. Ainsi, dans la classe `Point` il n'est pas nécessaire de définir la méthode `__clone()`, puisque tous ses attributs d'instance sont de type scalaire. En revanche il est nécessaire de la définir dans la classe `Segment` comme suit :

```
public function __clone()
{
    $this->start = clone $this->start;
    $this->end = clone $this->end;
}
```

Testez le clonage sur un point et un segment et vérifiez leur bon fonctionnement.

Modifiez la définition des constructeur, accesseur et modificateur dans la classe `Segment` en conséquence tout en préservant les solutions initialement proposées. Il vous suffira de les mettre en commentaire.