LES BONNES PRATIQUES

DE LA POO

**Les bonnes pratiques du code en général**

* **D.R.Y** : *Don't Repeat Yourself* - Moins on se répète, mieux on se porte !
* **K.I.S.S** : *Keep It Simple Stupid* - Faire toujours au plus simple !
* **Y.A.G.N.I** : *You Aren't Gonna Need It* - Écrire le code uniquement quand on en a besoin !
* **Tell, Don’t Ask** : Il vaut mieux que le code manipulant des données soit dans une même classe plutôt que dans deux classes séparées avec des getters.
* **Loi de Demeter** : Il vaut mieux dépendre de classes similaires à soi-même et ne pas dépendre de classes étrangères. Chaque unité ne doit parler qu’à ses amis. Ne pas parler aux étrangers.

**Comment savoir si je dois utiliser l’héritage ou la composition ?**

|  |  |  |
| --- | --- | --- |
|  | **MOT CLÉS** | **VERBE** |
| **COMPOSITION** | **NEW** | **AVOIR / UTILISER** |
| **HERITAGE** | **EXTENDS** | **ÊTRE** |

**Exemple :**

**Parrot extends Bird** oui un perroquet est un oiseau

**Bird extends Animal** oui un oiseau est un animal

**Parrot extends Feather** non car un perroquet n’est pas une plume, un perroquet a des plumes, il faudra donc utiliser la composition dans ce cas.

**Parrot extends Beak** non car un perroquet n’est pas un bec, un perroquet a un bec, il faudra donc utiliser la composition dans ce cas.

**La composition crée une dépendance forte avec la classe, dans certain cas il est préférable d’utiliser un design pattern nommé l’injection de dépendance.**

Référence au **T** de **STUPID** (Tight coupling - couplage fort).

## Les différences entre include et require et \_once

Les 4 fonctions (include, include\_once, require, require\_once) permettent d’inclure un fichier php.

Dans le cas d’un **include** si le fichier n’est pas accessible, problème de droit ou fichier inexistant, l’erreur qui se déclenche sera de type **Warning** la suite du code sera donc exécutée.

Dans le cas d’un **require** si le fichier n’est pas accessible, problème de droit ou fichier inexistant, l’erreur qui se déclenche sera de type **Error Fatal** php arrête directement le processus.

Dans le cas d’un **include\_once** php vérifiera si le fichier n’a pas déjà été chargé, et ne le chargera pas une seconde fois, si le fichier n’est pas accessible l’erreur sera également de type **Warning** comme dans le cas d’un **include**.

Dans le cas d’un **require\_once** php vérifiera si le fichier n’a pas déjà été chargé, et ne le chargera pas une seconde fois, si le fichier n’est pas accessible l’erreur sera également de type **Error Fatal** comme dans le cas d’un **require**.

**Concernant les interfaces :**

Une interface permet 2 choses, définir des contrats (imposer la déclaration de méthode aux classes implémentant l’interface), une interface est toujours abstraite elle doit représenter un groupe (une catégorie d’objet). Une interface permet de faire du polymorphisme, effectivement une classe implémentant une interface prendra le type d’instance de cette interface.

Une interface ne peut pas contenir de méthode avec des instructions, elle peut contenir uniquement des déclarations de contrats, représentant le prototype de la méthode).

|  |  |  |  |
| --- | --- | --- | --- |
| une interface | PEUT | extends | une autre interface |
| une interface | NE PEUT PAS | implements | une autre interface |
| une interface | NE PEUT PAS | extends | une classe |
| une interface | NE PEUT PAS | implements | une classe |
| une interface | NE PEUT PAS | être instanciée | |

Lorsque vous implémentez une interface vous avez l’obligation de déclarer les méthodes imposées par le contrat en respectant le prototype (nom, arguments et typage des arguments, typage de sortie (return)), seule les classes abstraites n’ont pas l’obligation de respecter les contrats de ce fait elles imposent ces déclarations à leurs héritiers (classe enfant) .

**Exemple d’utilisation des interfaces**

|  |
| --- |
| 1. <?php 2. interface Book { 3. public function getNumberPages(); 4. } 5. interface Dictionary extends Book { 6. public function firstWordInPage(integer $numPage); 7. } 8. interface TranslationDictionary extends Dictionary { 9. public function getCurrentLanguage(); 10. } 11. interface Novel extends Book { 12. public function getNumberChapter(); 13. } 14. interface Manga extends Book { 15. public function getNbPictures(); 16. } |

**Concernant les classes abstraites :** Une classe abstraite ne peut pas être instanciée, elle n’est utilisable que par héritage (extends), une classe abstraite n’a pas obligatoirement de méthodes abstraites, cependant si une classe à une méthode abstraite elle doit obligatoirement devenir abstraite, les méthodes abstraites sont des contrats et les héritiers (classes enfants) devront déclarer ces classes en respectant leur prototype. Une classe abstraites peut contenir des attributs et des méthodes avec des instructions.

|  |  |  |  |
| --- | --- | --- | --- |
| une classe abstraite | PEUT | extends | une autre classe abstraite |
| une classe abstraite | PEUT | extends | une classe (mais pas logique) |
| une classe abstraite | PEUT | implements | une interface |
| une classe abstraite | NE PEUT PAS | implements | une classe |
| une classe abstraite | NE PEUT PAS | extends | une interface |
| une classe abstraite | NE PEUT PAS | être instancié | |

**Exemple de classe abstraite (les commentaires donnent des informations sur les classes abstraites).**

|  |
| --- |
| 1. <?php 2. // Une classe abstraite peut être uniquement héritée et ne peut pas être instanciée, il n’est donc pas possible de faire new Shape. 3. abstract class Shape 4. { 5. // Déclarer un attribut color en visibilité protected 6. protected $sColor; 7. // Déclarer un attribut location en visibilité protected 8. protected $oLocation; 9. // Déclarer un attribut opacity en visibilité protected 10. protected $fOpacity; 12. public function \_\_construct() 13. { 14. $this->sColor = 'black'; 15. $this->oLocation = new Point(); // composition 16. $this->fOpacity = 1; 17. } 19. // Si une méthode est abstraite dans une classe, la classe doit être 20. // /!\ OBLIGATOIREMENT abstraite 21. // 22. // Une classe abstraite n'a pas obligatoirement de méthode abstraite 23. // 24. // Quand il y a une méthode abstraite, l'enfant héritant de la classe 25. // abstraite doit obligatoirement déclarer cette méthode en respectant les 26. // différents critères (visibilité, noms des arguments, typage des arguments, typage de sorties) 28. abstract public function draw(); 30. // Créer 2 Setters (setColor, setOpacity) 31. public function setColor($sColor) 32. { 33. $this->sColor = $sColor; 34. } 36. public function setOpacity($fOpacity) 37. { 38. $this->fOpacity = $fOpacity; 39. } 41. // setter (mutateur) son nom est setLocation il reçoit 2 parametres (x, y) 42. // x et y seront utilisés pour placer le Point 43. // (devra utiliser la méthode moveTo se trouvant dans Point) 44. public function setLocation($iX,$iY) 45. { 46. $this->oLocation->moveTo($iX, $iY); 47. } 48. } |

### Static

Un attribut déclaré **static** existe dès que sa classe est chargée en mémoire, en dehors et indépendamment de toute instanciation. Quel que soit le nombre d'instanciations de la classe (0, 1, ou plus) un attribut de classe, existe en un et un seul exemplaire. Un tel attribut sera utilisé un peu comme une variable globale d'un programme non objet.

Une méthode déclarée **static** existe également dès que sa classe est chargée en mémoire, en dehors et indépendamment de toute instanciation. Une méthode static pourra être utilisée un peu comme une fonction globale d'un programme non objet (il suffira de préfixer par le nom de la classe).

Nous comprenons à travers ces explications que le static nous ramène vers une programmation procédurale.

|  |
| --- |
| 1. <?php 2. class Test { 3. static $attribut = 1; 5. static public function testA() { 6. echo 'ok'; 7. } 9. public function testB() { 10. // il est possible d’appeler une méthode static depuis l’objet 11. $this->testA(); 12. // ou self::testA(); 13. // ou Test::testA(); 14. } 16. public function testC() { 17. echo Test::$attribut; 18. // ou Test::$attribut 19. // mais $this->attribut; ou $this->$attribut; ne sont pas possible 20. } 21. } 22. Test::testA(); 23. $o = new Test; 24. $o->testB(); 25. $o->testC(); |

### **Abstraction**

Les méthodes abstraites sont des contrats, si une classe étend une classe abstraite elle devra respecter tous les contrats (et donc devra contenir les différentes méthodes imposées).

Une classe abstraite pure (qui contient uniquement des méthodes abstraites) peut être modifiée en interface et doit être implémentée (implements) et non étendue (extends).

Pour rappel une classe abstraite n'a pas obligatoirement de méthode abstraite, cependant si une méthode est abstraite la classe doit être obligatoirement abstraite.

Il est impossible d'instancier une classe abstraite.

**Exemple d’utilisation d’abstraction:**

|  |
| --- |
| 1. <?php 2. class StandardEngine 3. { 4. private $capacity; 5. private $category; 6. } 7. class HybridEngine 8. { 9. private $capacity; 10. private $category; 11. } 12. abstract class Vehicule 13. { 14. abstract public function start($speed); 15. abstract public function stop(); 16. } 17. abstract class MotorizedVehicule extends Vehicule 18. { 19. protected $engine; 20. public function \_\_construct($engine) 21. { 22. $this->engine = $engine; 23. } 24. } 25. class Bicycle extends Vehicule 26. { 27. public function start($speed) 28. { 29. // Implémentation spécifique du démarrage du vélo 30. } 31. public function stop() 32. { 33. // Implémentation spécifique de l'arrêt du vélo 34. } 35. } 36. class Car extends MotorizedVehicule 37. { 38. private $color; 39. public function \_\_construct($engine, $color) 40. { 41. parent::\_\_construct($engine); // appelle le constructeur du parent 42. $this->setColor($color); 43. } 44. public function getColor() 45. { 46. return $this->color; 47. } 48. public function setColor($color) 49. { 50. if(in\_array($color, [ 'red', 'blue', 'silver']) == true) 51. { 52. $this->color = $color; 53. return; 54. } 55. throw new InvalidArgumentException('Cette couleur est impossible !'); 56. } 57. public function start($speed) 58. { 59. // Implémentation spécifique du démarrage de la voiture 60. } 61. public function stop() 62. { 63. // Implémentation spécifique de l'arrêt de la voiture 64. } 65. } 66. class Truck extends MotorizedVehicule 67. { 68. public function start($speed) 69. { 70. // Implémentation spécifique du démarrage du camion 71. } 72. public function stop() 73. { 74. // Implémentation spécifique de l'arrêt du camion 75. } 76. } 77. //////////////////////////// CODE CLIENT 78. $car1 = new Car(new HybridEngine(), 'red'); 79. $vehicule = new Truck(new StandardEngine()); 80. // ... beaucoup plus loin dans le code ... 81. $vehicule->start(50); |