Chapitre 4
Écrire des fonctions et des classes PHP

1. Fonctions

1.1 Introduction

À l'instar des différents langages de développement, PHP offre la possibilité de définir ses propres fonctions (appelées fonctions "utilisateur") avec tous les avantages associés (modularité, capitalisation...). Une fonction est un ensemble d'instructions identifiées par un nom, dont l'exécution retourne une valeur et dont l'appel peut être utilisé comme opérande dans une expression. Une procédure est un ensemble d'instructions identifiées par un nom qui peut être appelé comme une instruction.

1.2 Déclaration et appel

Le mot-clé `function` permet d'introduire la définition d'une fonction.

Syntaxe

```
function nom_fonction([paramètre]) [: type]{
  instructions;
}
```

`nom_fonction` Nom de la fonction (doit respecter les règles de nommage présentées dans le chapitre Introduction à PHP - Structure de base d'une page PHP). Ce nom n'est pas sensible à la casse (pour PHP, les fonctions `unefonction` et `UneFonction` sont les mêmes).

`paramètre`	Paramètres éventuels de la fonction exprimés sous forme d'une liste de variables (cf. section Fonctions - Paramètres) : `$paramètre1, $paramètre2, ...`
`type`	Déclaration du type de données retourné par la fonction. Valeurs possibles : `int`, `float`, `string`, `bool`, `array`, `callable`, `iterable`, `object`, `mixed`, `void`, un nom de classe ou d'interface (cf. dans ce chapitre la section Classes), ou une union de types. Le nom du type peut être précédé d'un point d'interrogation (`?`) qui indique que la fonction peut retourner une valeur `NULL`. Voir le chapitre Les bases du langage PHP pour la définition des types de données (section Les bases du langage PHP - Types de données).
`instructions`	Ensemble des instructions qui composent la fonction.

Le nom de la fonction ne doit pas être un mot réservé PHP (nom de fonction native, d'instruction) ni être égal au nom d'une autre fonction préalablement définie.

Une fonction utilisateur peut être appelée comme une fonction native de PHP : dans une affectation, dans une comparaison, etc.

Si la fonction retourne une valeur, il est possible d'utiliser l'instruction `return` pour définir la valeur de retour de la fonction.

Syntaxe

```
return expression;
```

`expression`	Expression dont le résultat constitue la valeur de retour de la fonction (`NULL` par défaut).

Le résultat d'une fonction peut être de n'importe quel type (chaîne, nombre, tableau, etc.).

L'instruction `return` stoppe l'exécution de la fonction et retourne le résultat de `expression` à l'appelant. Si plusieurs instructions `return` sont présentes dans la fonction, c'est la première rencontrée dans le déroulement des instructions qui définit la valeur de retour et provoque l'interruption de la fonction. Si la fonction ne comporte aucune instruction `return` (ou si aucune instruction `return` n'est exécutée), la valeur de retour de la fonction est `NULL`.

© Editions ENI - All rights reserved

Exemple

```
<?php
// Fonction sans paramètre qui affiche "Bonjour !"
// Pas de valeur de retour.
function afficher_bonjour() {
  echo 'Bonjour !<br />';
}
// Fonction avec deux paramètres qui retourne le produit
// des deux paramètres.
function produit($valeur1,$valeur2) {
  return $valeur1 * $valeur2;
}
// Appel de la fonction afficher_bonjour.
afficher_bonjour();
// Utilisations de la fonction produit :
// - dans une affectation
$résultat = produit(2,4);
echo "2 x 4 = $résultat<br />";
// - dans une comparaison
if (produit(10,12) > 100) {
  echo '10 x 12 est supérieur à 100.<br />';
}
?>
```

Résultat

```
Bonjour !
2 x 4 = 8
10 x 12 est supérieur à 100.
```

Remarque

Dans le langage PHP, il n'existe pas à proprement parler de procédure. Pour définir quelque chose d'équivalent à une procédure, il suffit de définir une fonction qui ne retourne pas de valeur et d'appeler la fonction comme s'il s'agissait d'une instruction (comme la fonction `afficher_bonjour` *par exemple). Une fonction qui ne retourne rien peut explicitement être déclarée avec le type de retour* `void`*.*

Comme nous l'avons déjà évoqué, le contenu d'un tableau peut être transformé en liste de paramètres dans un appel de fonction grâce à l'opérateur `...` (points de suspension).

Exemple

```
<?php
// Fonction avec trois paramètres qui retourne la somme
// des trois paramètres.
function somme($valeur1,$valeur2,$valeur3) {
  return $valeur1 + $valeur2 + $valeur3;
}
// Transformation du contenu d'un tableau en
// liste de paramètres.
$valeurs = [1,2,3];
echo '1 + 2 + 3 = ',somme(...$valeurs),'<br />';
// La même chose pour une partie seulement des paramètres
// avec un tableau défini directement dans l'appel.
echo '1 + 2 + 4 = ',somme(1,...[2,4]),'<br />';
?>
```

Résultat

```
1 + 2 + 3 = 6
1 + 2 + 4 = 7
```

Lorsqu'une fonction retourne un tableau, il est possible d'accéder directement à un élément du tableau lors de l'appel à la fonction avec une syntaxe du type `fonction(...)[clé]`.

Exemple

```
<?php
// Définition d'une fonction qui retourne un tableau.
function qui() {
  return ['Olivier','Heurtel'];
}
// Appel de la fonction et récupération directe du prénom stocké
// à l'indice 0 du tableau retourné.
$prénom = qui()[0];
echo "qui()[0] = $prénom<br />";
?>
```

Résultat

```
qui()[0] = Olivier
```

Cette technique fonctionne aussi lorsque la fonction retourne un tableau multidimensionnel avec une syntaxe du type `fonction(...)[clé1][clé2]`.

Il est possible d'utiliser une fonction avant de la définir.

© Editions ENI - All rights reserved

Exemple

```
<?php
// Utilisation de la fonction produit.
echo produit(5,5);
// Définition de la fonction produit.
function produit($valeur1,$valeur2) {
  return $valeur1 * $valeur2;
}
?>
```

Résultat

```
25
```

Il n'y a donc aucun problème pour définir des fonctions qui s'appellent entre elles.

Remarque

Une fonction est utilisable uniquement dans le script où elle est définie. Pour l'employer dans plusieurs scripts, il faut, soit recopier sa définition dans les différents scripts (vous perdez l'intérêt de définir une fonction), soit la définir dans un fichier inclus partout où la fonction est nécessaire.

Exemple

– Fichier `fonctions.inc` contenant des définitions de fonctions :

```
<?php
// Définition de la fonction produit.
function produit($valeur1,$valeur2) {
  return $valeur1 * $valeur2;
}
?>
```

– Script utilisant les fonctions définies dans `fonctions.inc` :

```
<?php
// Inclusion du fichier contenant la définition des fonctions.
include('fonctions.inc');
// Utilisation de la fonction produit.
echo produit(5,5);
?>
```

Déclaration du type de retour

Il est possible de définir le type de données retourné par une fonction.

Lorsque c'est le cas, dans le mode de fonctionnement par défaut (à opposer au mode strict présenté ci-dessous), PHP effectue si besoin une conversion automatique de la valeur retournée dans le type de données déclaré.

Exemple

```
<?php
// Déclaration de deux fonctions qui retournent le produit
// des deux paramètres, la deuxième spécifiant un type
// de données "entier" pour la valeur de retour.
function produit1($valeur1,$valeur2) {
  return $valeur1 * $valeur2;
}
function produit2($valeur1,$valeur2) : int {
  return $valeur1 * $valeur2;
}
// Appel des deux fonctions avec les mêmes paramètres
echo 'produit1(20,1/7) => ',var_dump(produit1(20,1/7)),'<br />';
echo 'produit2(20,1/7) => <b>',var_dump(produit2(20,1/7)),'</b><br />';
?>
```

Résultat

```
produit1(20,1/7) => float(2.8571428571428568)
produit2(20,1/7) => int(2)
```

Sur cet exemple, nous voyons bien que la valeur retournée par la deuxième fonction a été convertie en entier par PHP (avec les règles de conversion évoquées dans le chapitre Introduction à PHP - Les bases du langage PHP - Types de données).

Si PHP n'est pas en mesure d'effectuer la conversion (types de données non convertibles entre eux), une exception `TypeError` est levée ; cette exception interrompt le script si elle n'est pas gérée (cf. dans ce chapitre la section Classes - Exceptions).

Exemple

```
<?php
// Déclaration et appel d'une fonction qui doit retourner un
// tableau mais qui retourne une chaîne de caractères.
function qui() : array {
  return 'Olivier Heurtel';
}
echo 'qui()[0] = ',qui()[0];
?>
```

Résultat

```
qui()[0] =
Fatal error: Uncaught TypeError: Return value of qui() must be of the
type array, string returned in /app/scripts/index.php:5 Stack trace: #0
/ app/scripts/index.php(7): qui() #1 {main} thrown in/app/scripts/
index.php on
line 5
```

© Editions ENI - All rights reserved

Une fonction déclarée avec un type de retour autre que `void` doit retourner une valeur non `NULL`. Si ce n'est pas le cas, une erreur est retournée, différente selon les cas :

Absence d'instruction `return`

```
Fatal error: Uncaught TypeError: Return value of MaFonction() must be of
the type int, none returned in ...
```

Instruction `return` vide

```
Fatal error: A function with return type must return a value in ...
```

Instruction `return NULL`

```
Fatal error: Uncaught TypeError: Return value of MaFonction() must be of
type int, null returned in ...
```

Pour autoriser une fonction à retourner une valeur `NULL`, il faut faire précéder le nom du type (autre que `void`) d'un point d'interrogation (`?`).

```
<?php
// Déclaration et appel d'une fonction qui spécifie un
// type de donnée de retour qui peut être NULL
function cube($valeur) : ?int {
  if (is_null($valeur)) {
    return NULL;
  } else {
    return $valeur ** 3 ;
  }
}
echo 'cube(2) => <b>',var_dump(cube(2)),'</b><br />';
echo 'cube(NULL) => <b>',var_dump(cube(NULL)),'</b><br />';
?>
```

Résultat

```
cube(2) => int(8)
cube(NULL) => NULL
```

Même avec cette option, la fonction doit avoir une instruction `return` non vide. Si ce n'est pas le cas, une erreur est retournée, différente selon les cas :

Absence d'instruction `return`

```
Fatal error: Uncaught TypeError: Return value of MaFonction() must be of
type ?int, none returned in ...
```

Instruction `return` vide

```
Fatal error: A function with return type must return a value (did you mean
"return null;" instead of "return;"?) in ...
```

À l'inverse, il est possible de déclarer qu'une fonction ne retourne rien, en utilisant `void` comme nom de type. Dans ce cas, la fonction doit omettre l'instruction `return` ou mettre une instruction `return` vide, sans valeur (même pas `NULL`).

Chapitre 4
Le design pattern Abstract Factory

1. Description

Le but du design pattern `Abstract Factory` est la création d'objets regroupés en familles sans avoir à connaître leurs classes concrètes.

2. Exemple

Le système de vente de véhicules gère des véhicules fonctionnant à l'essence et des véhicules fonctionnant à l'électricité. Cette gestion est confiée à l'objet `Catalogue`, à qui incombe la responsabilité de créer de tels objets.

Pour chaque produit, nous disposons d'une classe abstraite, d'une sous-classe concrète décrivant la version du produit fonctionnant à l'essence et d'une sous-classe concrète décrivant la version du produit fonctionnant à l'électricité. Par exemple, à la figure 4.1, pour l'objet scooter, il existe une classe abstraite `Scooter` et deux sous-classes concrètes : `ScooterElectricite` et `ScooterEssence`.

L'objet `Catalogue` peut utiliser ces sous-classes concrètes pour instancier les produits. Cependant, si par la suite de nouvelles familles de véhicules doivent être prises en compte (diesel ou hybride essence-électricité), les modifications à apporter à l'objet `Catalogue` peuvent s'avérer assez fastidieuses.

Le design pattern `Abstract Factory` résout ce problème en introduisant une interface `FabriqueVehiculeInterface` qui contient la signature des méthodes à utiliser pour créer chaque produit. Le type de retour de ces méthodes est constitué par l'une des classes abstraites de produit. Ainsi, l'objet `Catalogue` n'a pas besoin de connaître les sous-classes concrètes et reste parfaitement indépendant des familles de produits. En révélant l'interface de nos fabriques et non leur implémentation, nous découplons le code client des produits concrets : notre objet client `Catalogue` demande des produits à la fabrique qu'on lui passe en paramètre lors de sa construction sans avoir la moindre idée de qui elle est ni de ce qui se passe en coulisses pour qu'il obtienne le bon produit.

Une classe implémentant `FabriqueVehiculeInterface` est créée pour chaque famille de produits, à savoir les classes `FabriqueVehiculeElectricite` et `FabriqueVehiculeEssence`. Une telle classe a la responsabilité d'implémenter les opérations de création du véhicule appropriée pour la famille à laquelle elle est associée.

L'objet client `Catalogue` prend alors pour paramètre un objet se conformant à l'interface `FabriqueVehiculeInterface`, c'est-à-dire soit une instance de `FabriqueVehiculeElectricite`, soit une instance de `FabriqueVehiculeEssence`. Avec une telle instance, le catalogue peut créer et manipuler des véhicules sans devoir connaître les familles de véhicules et les classes concrètes correspondantes.

© Editions ENI - All rights reserved

L'ensemble des classes du design pattern `Abstract Factory` pour cet exemple est détaillé à la figure 4.1.

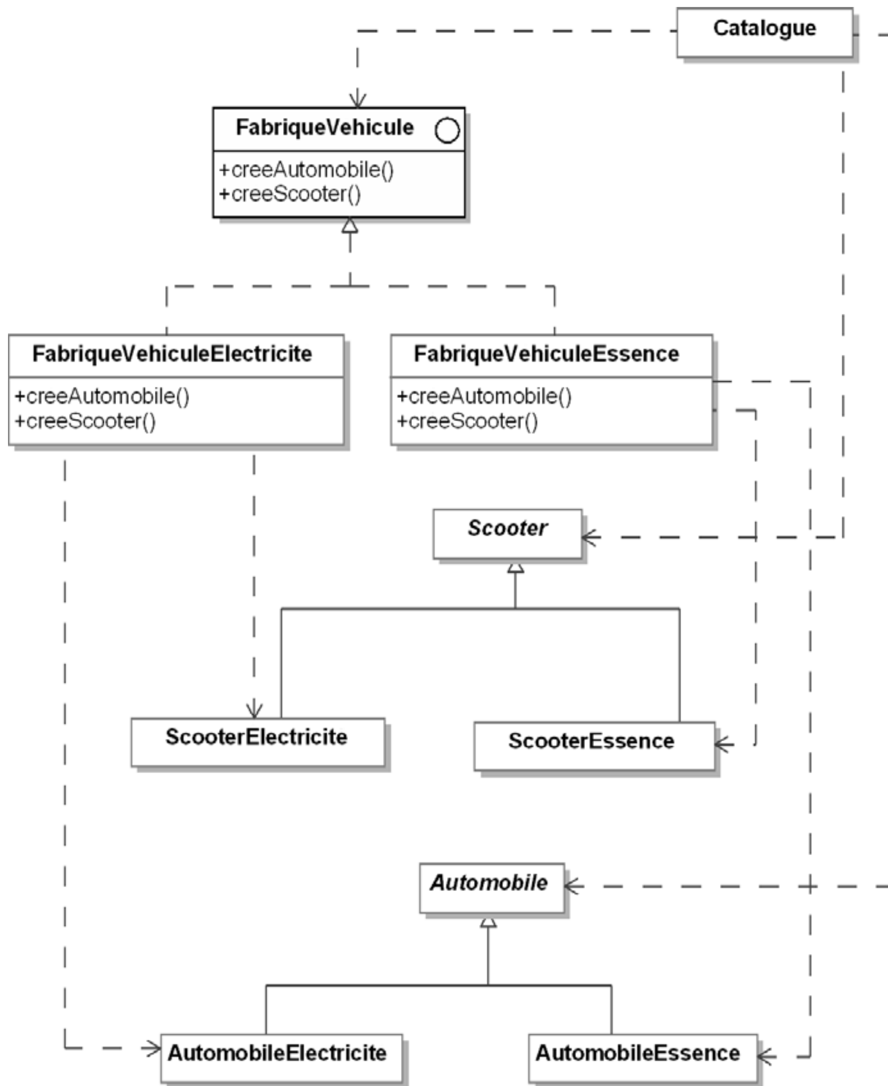

Figure 4.1 - Le design pattern `Abstract Factory` appliqué à des familles de véhicules

3. Structure

3.1 Diagramme de classes

La figure 4.2 détaille la structure générique du design pattern `Abstract Factory`.

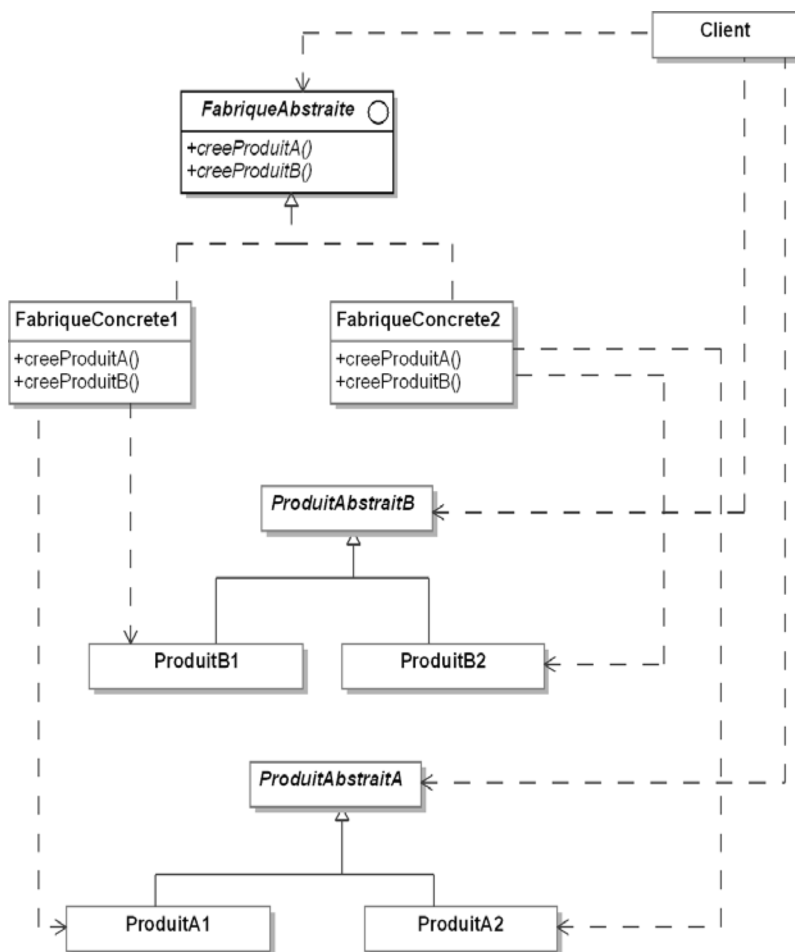

Figure 4.2 - Structure du design pattern `Abstract Factory`

© Editions ENI - All rights reserved

3.2 Participants

Les participants au design pattern `Abstract Factory` sont les suivants :

- `FabriqueAbstraite` (`FabriqueVehiculeInterface`) est une interface spécifiant les signatures des méthodes créant les différents produits.
- `FabriqueConcrete1`, `FabriqueConcrete2` (`FabriqueVehiculeElectricite`, `FabriqueVehiculeEssence`) sont les classes concrètes implémentant les méthodes créant les produits pour chaque famille de produits. Connaissant la famille et le produit, elles sont capables de créer une instance du produit pour cette famille.
- `ProduitAbstraitA` et `ProduitAbstraitB` (`AbstractScooter` et `AbstractAutomobile`) sont les classes abstraites des produits indépendamment de leur famille. Les familles sont introduites dans leurs sous-classes concrètes.
- `Client` (`Catalogue`) est la classe qui utilise l'interface `FabriqueAbstraite`.

3.3 Collaborations

La classe `Catalogue` utilise une instance de l'une des fabriques concrètes pour créer ses produits au travers de l'interface exposée par `FabriqueAbstraiteInterface`.

Remarque

Il est recommandé de ne créer qu'une seule instance des fabriques concrètes, celle-ci pouvant être partagée par plusieurs clients. Nous verrons plus tard un design pattern capable de garantir qu'une seule instance d'une classe est disponible à l'exécution : `Singleton`.

4. Domaines d'utilisation

Le design pattern `Abstract Factory` est utilisé dans les domaines suivants :

- Un système utilisant des produits a besoin d'être indépendant de la façon dont ces produits sont créés et regroupés.
- Un système est paramétré par plusieurs familles de produits qui peuvent évoluer.

5. Exemple en PHP

Voici maintenant un exemple d'utilisation du design pattern écrit en PHP. Le code PHP correspondant à la classe abstraite `AbstractAutomobile` et ses sous-classes est donné à la suite. Il est très simple, il décrit les quatre propriétés des automobiles ainsi que la méthode `afficheCaracteristiques` qui permet de les afficher.

```
<?php

declare(strict_types=1);

namespace ENI\DesignPatterns\AbstractFactory;

abstract class AbstractAutomobile
{
   protected string $marque;

   protected string $couleur;

   protected int $puissance;

   protected float $espace;

   public function __construct(string $marque, string $couleur,
int $puissance, float $espace)
   {
       $this->marque = $marque;
       $this->couleur = $couleur;
       $this->puissance = $puissance;
```

© Editions ENI - All rights reserved

```
        $this->espace = $espace;
    }

    abstract public function afficheCaracteristiques(): void;
}


<?php

declare(strict_types=1);

namespace ENI\DesignPatterns\AbstractFactory;

class AutomobileElectricite extends AbstractAutomobile
{
    public function afficheCaracteristiques(): void
    {
        echo "Automobile électrique - marque: $this->marque"
            . ", couleur: $this->couleur"
            . ", puissance: $this->puissance"
            . ", espace: $this->espace" . PHP_EOL;
    }
}


<?php

declare(strict_types=1);

namespace ENI\DesignPatterns\AbstractFactory;

class AutomobileEssence extends AbstractAutomobile
{
    public function afficheCaracteristiques(): void
    {
        echo "Automobile à essence - marque: $this->marque"
            . ", couleur: $this->couleur"
            . ", puissance: $this->puissance"
            . ", espace: $this->espace" . PHP_EOL;
    }
}
```

Le code PHP correspondant à la classe abstraite `AbstractScooter` et ses sous-classes est donné à la suite. Il est similaire à celui des automobiles, à