

La Programmation Orientée

Objet : design patterns

Table des matières

I. Contexte	3
II. Le design pattern Singleton	3
III. Exercice : Appliquez la notion	9
IV. Le design pattern Factory	9
V. Exercice : Appliquez la notion	12
VI. Autres design patterns	13
VII. Exercice : Appliquez la notion	17
VIII. Auto-évaluation	18
A. Exercice final	18
B. Exercice : Défi	20
Solutions des exercices	21

I. Contexte

Durée : 1 h

Environnement de travail : Local

Pré-requis : Connaître les bases de PHP et avoir installé et configuré un serveur de test local

Contexte

Les design patterns, ou patrons de conception, sont des méthodes de conceptions standardisées et généralistes visant à répondre à des besoins récurrents.

Par essence, un design pattern est une bonne pratique. Il peut arriver avec le temps et les évolutions qu'un design pattern finisse par être considéré comme une mauvaise pratique, on parle alors d'anti-pattern.

Leur but sera alors d'apporter une réponse simple à un problème commun, qui sera efficiente, facile à développer et à maintenir.

II. Le design pattern Singleton

Objectifs

- Découvrir le design pattern Singleton
- Analyser son implémentation et comment l'utiliser
- Voir ses limites et son alternative

Mise en situation

Pendant le développement d'une application, il se peut que nous nous retrouvions face à des fonctionnalités assez lourdes qui nécessitent une quantité de mémoire élevée.

Face à ce constat, un design pattern peut répondre à notre besoin : le Singleton.

Définition Le principe : une instance unique

Un Singleton est une classe qui ne peut être instanciée qu'une seule fois. On ne doit pas pouvoir faire de `new Singleton()` ni cloner un objet déjà créé.

On travaillera alors toujours avec le même objet, libérant de l'espace par rapport à un cas où nous en créerions un nouveau à chaque utilisation.

Méthode Mise en œuvre

Pour mettre en œuvre ce type de classe, il faut rendre privées ses méthodes `__construct` et `__clone`.

Il faut également stocker sa propre instance dans une propriété `static` de la classe.

Enfin, il est nécessaire de mettre en place une méthode qui l'instanciera seulement dans le cas où elle ne l'a pas déjà été, puis qui retournera son instance.

Exemple

Voici l'implémentation d'un Singleton qui permettrait d'écrire dans un fichier de log :

```

1 <?php
2
3 class Logger
4 {
5     // On stockera l'instance du Singleton dans la propriété instance */
6     private static $instance;
7
8     /**
9      * Le constructeur et la méthode de clonage doivent être privés afin de ne pas pouvoir
10     * instancier notre Singleton via un new Logger(); ni de cloner une instance existante
11     */
12     private function __construct() {}
13     private function __clone() {}
14
15     /**
16     * La méthode statique qui nous permet de générer l'instance unique de notre
17     * Singleton ou de la retourner si elle a déjà été créée
18     */
19     public static function getInstance(): self
20     {
21         // On vérifie que notre Singleton n'a pas déjà été instancié
22         if (!isset(static::$instance)) {
23             /**
24              * S'il n'a pas encore été instancié, alors on crée la nouvelle
25              * instance et on la stocke dans la propriété statique $instance
26              */
27             static::$instance = new static;
28         }
29
30         /**
31          * On retourne l'instance unique de notre Singleton
32          */
33         return static::$instance;
34     }
35
36     public function logToFile(string $data): void
37     {
38         // code métier de votre Singleton, ici on loggerait les données fournies
39     }
40 }
41
42 $logger1 = Logger::getInstance();
43 $logger2 = Logger::getInstance();
44
45 if ($logger1 === $logger2) {
46     echo "C'est bien le même objet instancié une seule fois";
47 }

```

self (: self ou self::)

Les références statiques avec self sont résolues en utilisant la classe à laquelle appartiennent les fonctions, celle où elles ont été définies.

static (static::) ou résolution statique à la volée

Les références statiques avec static sont résolues en utilisant la classe qui a été appelée.

Exemple

```
1 <?php
2 class A {
3     public static function qui() {
4         echo "classe A";
5     }
6     public static function test() {
7         self::qui();
8     }
9 }
10
11 class B extends A {
12     public static function qui() {
13         echo "classe B";
14     }
15 }
16
17 B::test(); // retourne "classe A"
18
19 class C {
20     public static function qui() {
21         echo "classe C";
22     }
23     public static function test() {
24         static::qui(); // Ici, résolution à la volée
25     }
26 }
27
28 class D extends C {
29     public static function qui() {
30         echo "classe D";
31     }
32 }
33 D::test(); // retourne "classe D"
34 ?>
```

void (: void)

Une fonction de type void ne retourne aucune valeur, elle se termine soit par une déclaration soit retour vide (return;) soit sans déclaration de retour.

Avantages du Singleton

Comme nous l'avons évoqué, l'avantage principal du design pattern Singleton est de libérer de la mémoire par l'instanciation d'un objet unique.

Les opérations lourdes peuvent alors être effectuées une seule et unique fois.

La connexion/déconnexion à la base de données est un exemple typique d'utilisation de ce patron de conception.

Exemple

Voici une utilisation du design pattern qui retourne une instance unique de la classe PDO, permettant de communiquer avec une base de données :

```

1 <?php
2
3 class PDOSingleton
4 {
5     private static PDO $instance;
6
7     private function __construct() {}
8     private function __clone() {}
9
10    public static function getInstance(): PDO
11    {
12        if (!isset(self::$instance)) {
13            self::$instance = new PDO('mysql:host=localhost;dbname=dbname;charset=utf8',
14            'root', '');
15        }
16
17        return self::$instance;
18    }
19 }
20 $pdo = PDOSingleton::getInstance();

```

Un autre avantage inhérent aux Singletons est la possibilité de les instancier de n'importe où.

Pour reprendre l'exemple de PDO Singleton, nous pourrions avoir accès à la même instance de PDO dans tous nos modèles, d'un simple appel à la méthode statique `getInstance()`.

Attention

Un patron de conception à utiliser avec parcimonie !

Si le Singleton peut s'avérer utile et efficace, comme c'est le cas pour PDO, son utilisation doit être mûrement réfléchie.

En effet, si ce patron de conception est si connu, c'est aussi parce qu'il est très souvent utilisé à mauvais escient. Cela lui vaut d'être parfois considéré comme un anti-pattern.

Le problème principal de ce design est le très fort couplage entre les classes que cela va instaurer.

Cela peut rendre le code difficile à faire évoluer et maintenir, mais aussi à tester unitairement.

Une alternative : l'injection de dépendance

Comme nous l'avons vu, le but d'un patron de conception est de répondre à un besoin récurrent de façon optimale.

Puisque les Singletons avaient des limites et ne devaient pas être utilisés dans toutes les situations, il était alors logique qu'un autre design pattern naisse afin de répondre à ce problème de fort couplage : l'injection de dépendance.

Exemple

Partons d'une application qui gère des utilisateurs, ces utilisateurs ont chacun une adresse, cette adresse est une composante de plusieurs variables. Le développement sans patron de conception commencerait de la sorte.

```

1 class Address
2 {
3     private $number;
4     private $street;
5     private $zipcode;
6     private $city;
7
8     public function __construct($number, $street, $zipcode, $city)
9     {
10         $this->number = $number;
11         $this->street = $street;
12         $this->zipcode = $zipcode;
13         $this->city = $city;
14     }
15 }
16
17 class Person
18 {
19     private $address;
20
21     public function __construct($number, $street, $zipcode, $city)
22     {
23         $this->address = new Address($number, $street, $zipcode, $city);
24     }
25 }
26
27 $person = new Person(4, 'chemin du village', 11110, 'Narbonne city');
```

Le problème est que la classe Person est étroitement liée à la classe Address, la classe personne est même inutilisable sans la classe adresse. De plus, imaginons que l'on souhaite ajouter une variable à la classe Address, par exemple \$country, il faudra aussi modifier le constructeur de la classe Person et lui rajouter un paramètre. La solution pour éviter toutes ses manipulations et l'injection de dépendance.

```

1 class Person
2 {
3     private $address;
4
5     public function __construct(Address $address)
6     {
7         $this->address = $address;
8     }
9 }
```

Si une de vos classes a besoin d'une instance d'une autre classe dans une de ses méthodes ou son constructeur alors elle prendra cette instance directement en paramètre et ne se chargera pas de l'instancier elle-même.

Les interfaces en PHP

Les interfaces permettent de créer un modèle que les classes qui l'implémentent doivent respecter. Il faut voir l'interface comme un contrat d'utilisation. En implémentant une interface, une classe s'oblige à définir l'ensemble de ses méthodes.

```

1 <?php
2 interface IMonInterfaceStatic
3 {
4     static function staticFonc1($name);
5     static function staticFonc2($firstName);
6 }
7
8 class MaClasse implements IMonInterface
9 {
10     public function Fonc1($name);
11     {
12         echo $name;
13     }
14     public function fonc2($firstName)
15     {
16         echo $firstName;
17     }
18 }
19 ?>

```

Si une classe ne redéfinit pas toutes les méthodes d'une interface quelle implémente, cela produit une erreur. De plus, une interface ne peut pas lister des méthodes privées, abstraites ou finales. En utilisant le principe d'injection de dépendance il est possible d'injecter des interfaces.

Toujours avec une connexion à une base de données, avec ce design, il est par exemple possible d'injecter une interface qui sera par la suite implémentée afin de créer différents types de connexions.

```

1 <?php
2
3 interface ConnectionInterface
4 {
5     public function getResult(string $query): array;
6 }
7
8 class Manager
9 {
10     private ConnectionInterface $connection;
11
12     public function __construct(ConnectionInterface $connection)
13     {
14         $this->connection = $connection;
15     }
16
17     public function query(string $query): array
18     {
19         return $this->connection->getResult($query);
20     }
21 }

```

De cette façon, le code est totalement découplé.

Pour créer différents types de connexions, il suffit de créer autant de classes implémentant `ConnectionInterface` que nécessaire, avec les variations impliquées par chacune.

On peut ainsi utiliser PDO ou MySQLi sans que cela affecte notre manager, ou encore avoir différentes classes pour se connecter à différentes bases de données. Toutes les variations sont possibles, tant que l'interface est implémentée.

Il ne reste plus qu'à créer une instance de connexion et la passer au constructeur du manager :

```

1 <?php
2
3 class PDOConnection implements ConnectionInterface {
4 // code de la classe PDOConnection
5 }
6
7 class MySQLiConnection implements ConnectionInterface {
8 // code de la classe MySQLiConnection
9 }
10
11 // Une classe qui implémente ConnectionInterface et se base sur PDO
12 $PDO = new PDOConnection();
13 $pdoManager = new Manager($PDO);
14
15 // Une classe qui implémente ConnectionInterface et se base sur MySQLi
16 $MySQLi = new MySQLiConnection();
17 $mysqliManager = new Manager($MySQLi);

```

Syntaxe À retenir

- Un Singleton est une classe qui ne peut être instanciée qu'une fois, optimisant la consommation de mémoire d'un script.
- Attention à l'utilisation de ce design pattern, il est très spécifique et, mal utilisé, il est nuisible à la qualité du code, qui devient fortement couplé.
- Une alternative au Singleton est l'injection de dépendance.

III. Exercice : Appliquez la notion

Pour réaliser cet exercice, vous pouvez travailler sur l'environnement de travail :



Question

[solution n°1 p.23]

Mettez en place une classe `DatabaseManager` qui implémente les éléments essentiels de ce design pattern, à savoir :

- Une propriété `instance` permettant de stocker l'instance
- Un constructeur et une méthode de clonage privée
- Une méthode statique `getInstance` permettant d'instancier ou de retourner cette instance
- Une méthode métier `connect`

Vérifiez que votre classe ne peut être instanciée qu'une seule fois.

IV. Le design pattern Factory

1 <https://repl.it/>

Objectifs

- Découvrir le design pattern Factory
- Analyser son implémentation et son utilisation

Mise en situation

Dans un système applicatif, il existe des situations où une action ayant la même finalité doit être exécutée d'une façon ou d'une autre.

Prenons par exemple le cas d'une application qui, à intervalle régulier, notifierait ses utilisateurs de son état. Si tout se passe correctement, les utilisateurs sont notifiés de la situation par e-mail. Si un problème critique est détecté, ils sont notifiés via un autre canal de communication (Slack, un système de messagerie instantanée, par exemple).

Dans ces deux situations, la finalité reste la même : prévenir l'utilisateur en envoyant une notification. La logique de la méthode d'envoi sera en revanche différente selon le cas.

Un patron de conception a été pensé pour répondre à ces situations : le design pattern **Factory**.

Méthode Le principe : une classe "chef d'orchestre"

Une classe Factory est une classe dont le rôle est d'instancier d'autres classes pour une situation donnée. Grâce à ce type de classes, la logique d'instanciation d'un objet se retrouve déportée et sera toujours la même.

Exemple

Reprenons l'exemple évoqué ci-dessus.

Nous l'avons dit, selon le contexte, nous souhaitons faire varier le canal de communication qui sera utilisé pour notifier un utilisateur du statut de l'application.

L'utilisation d'un tel design dans ce cas de figure permettrait, qu'importe la situation, de toujours gérer une notification de la même façon.

```

1 <?php
2
3 abstract class Notification
4 {
5     // Tout type de notification doit implémenter une méthode send()
6     protected abstract function send(string $message);
7
8     public function manageNotification($message)
9     {
10         // Qu'importe le type de notification, on effectuera des actions préalables et on
11         // envoie un message
12         $this->doStuff();
13         $this->send($message);
14     }
15
16     public function doStuff()
17     {
18         // On pourrait logger des informations
19     }
20 }
21
22 class EmailNotification extends Notification
23 {
24     public $recipient;
25     public $subject;

```

```

25
26 public function __construct()
27 {
28     $this->recipient = "contact@societe.com";
29     $this->subject = 'Etat applicatif';
30 }
31
32 // La méthode send dans ce cas envoie un e-mail
33 protected function send(string $message)
34 {
35     echo sprintf('On envoie le mail ayant pour contenu "%s" au contact "%s" avec pour
36 sujet : %s <br/>', $message, $this->recipient, $this->subject);
37 }
38
39 class SlackNotification extends Notification
40 {
41     public $channel;
42
43     public function __construct()
44     {
45         $this->channel = "#applicationState";
46     }
47
48     public function doStuff()
49     {
50         parent::doStuff();
51         // do something else
52     }
53
54     // Ici on poste le message sur un canal
55     protected function send(string $message)
56     {
57         echo sprintf('On notifie le canal %s du message : %s <br/>', $this->channel,
58 $message);
59     }
60 }
61
62 class NotificationFactory
63 {
64     // Selon l'état, on décide de renvoyer un type de notification ou un autre
65     public static function createNotificationForState(string $applicationState)
66     {
67         switch ($applicationState) {
68             case 'problem':
69                 return new SlackNotification();
70             case 'normal':
71             default:
72                 return new EmailNotification();
73         }
74     }
75 }
76
77 $notification1 = NotificationFactory::createNotificationForState('problem');
78 // Affichera : On notifie le canal #applicationState du message : La base de données est
79 inaccessible

```

```

80 $notification1->manageNotification('La base de données est inaccessible');
81
82 $notification2 = NotificationFactory::createNotificationForState('normal');
83 // Affichera : On envoie le mail ayant pour contenu "Tout va bien" au contact
   "contact@societe.com" avec pour sujet : Etat applicatif
84 $notification2->manageNotification('Tout va bien');
85
86 // Qu'importe ce qui a pu être retourné, on effectue le travail de la même façon
87

```

Avantage : favoriser l'évolutivité

L'avantage principal de ce patron de conception est de rendre les applications plus extensibles, plus faciles à faire évoluer.

Dans l'exemple que nous venons de voir, si nous souhaitons mettre en place des notifications par SMS, nous n'aurions qu'à :

- Créer une classe `SMSNotification` qui étendrait la classe `Notification`,
- Modifier la fonction `createNotificationForState` de notre `Factory` pour implémenter ce nouveau système.

Le type de notification utilisé pour un statut donné pourrait également varier dans le temps sans que le script permettant d'envoyer les notifications ne soit impacté.

Syntaxe À retenir

- Une `Factory` est une classe qui permet d'instancier différentes implémentations d'une classe.
- Utiliser une `Factory` simplifie l'évolution du code. Il ne sera plus directement couplé à une seule et unique classe, mais à une classe qui devra respecter une certaine structure. Celle-ci sera retournée en fonction du contexte.
- La logique utilisée est libre et dépendra essentiellement du contexte de l'application.

V. Exercice : Appliquez la notion

Pour réaliser cet exercice, vous pouvez travailler sur l'environnement de travail :



Question

[solution n°2 p.23]

Créez 3 classes représentant les animaux suivants : `Cat` / `Dog` / `Horse`.

Ces classes implémenteront l'interface suivante afin de retourner le nom du cri de chaque animal.

```

1 <?php
2
3 interface AnimalInterface
4 {
5     public function getSoundType(): string;
6 }

```

1 <https://repl.it/>

Mettez en place une classe supplémentaire `AnimalFactory` implémentant le pattern Factory et permettant d'instancier et de retourner, pour un type d'animal donné, l'objet correspondant grâce à une méthode `load`.

Si l'on essaie d'instancier un type d'animal n'ayant pas été défini, une erreur sera déclenchée et sera correctement interprétée.

Voici les types d'animaux que vous devrez vérifier :

```
1 $animalTypes = ['horse', 'dog', 'mice', 'cat', 'lion'];
```

VI. Autres design patterns

Objectifs

- Découvrir le design pattern Observer
- Découvrir le design pattern Strategy

Mise en situation

Parmi les nombreux design pattern existants, deux autres sont souvent utilisés et méritent d'être abordés, les patrons de conception Observer et Strategy.

Méthode Le design pattern Observer

Le principe est simple : des classes qui "écoutent" d'autres classes.

Il y a des classes "Subject", qui sont écoutées, et des classes "Observer" qui sont celles qui les écoutent.

Le but est de pouvoir exécuter du code spécifique lors de la mise à jour d'une classe, sans avoir à modifier son comportement pour autant, mais simplement en lui ajoutant un Observer.

PHP fournit nativement deux interfaces qui permettent de mettre en place ce design pattern : les interfaces `SplSubject` et `SplObserver`.

```
1 <?php
2
3 interface SplSubject
4 {
5     // Permet, durant l'exécution de notre script, de rajouter un Observer à un objet
6     public function attach(SplObserver $observer);
7
8     // Permet, durant l'exécution de notre script, de retirer un Observer d'un objet
9     public function detach(SplObserver $observer);
10
11     // La méthode qui va "tenir au courant" les Observer
12     public function notify();
13 }
14
15 interface SplObserver
16 {
17     // La méthode à exécuter lorsque l'Observer est notifié
18     public function update(SplSubject $subject);
19 }
```

Exemple Implémentation de ce patron de conception en PHP

Prenons une classe `Personne`, et déclenchons des événements dès que l'ont fait appel à la méthode permettant de mettre à jour son âge.

```

1 <?php
2
3 class Person implements SplSubject
4 {
5     private $observers;
6
7     private $age;
8
9     public function __construct(int $age)
10    {
11        $this->observers = new SplObjectStorage();
12        $this->age = $age;
13    }
14
15    public function attach(SplObserver $observer)
16    {
17        $this->observers->attach($observer);
18    }
19
20    public function detach(SplObserver $observer)
21    {
22        $this->observers->detach($observer);
23    }
24
25    public function notify()
26    {
27        foreach ($this->observers as $observer) {
28            // Lors de la notification des Observer, nous lançons leurs méthodes à exécuter
29            dans cette situation.
30            $observer->update($this);
31        }
32    }
33
34    public function updateAge()
35    {
36        $this->age++;
37        $this->notify();
38
39        return $this;
40    }
41
42    public function getAge()
43    {
44        return $this->age;
45    }
46 }
47
48 class SayHappyBirthday implements SplObserver
49 {
50     public function update(SplSubject $person)
51     {
52         // Code métier à exécuter lorsque la classe est notifiée

```

```

53         echo sprintf('Joyeux anniversaire ! Vous avez désormais %s ans<br/>', $person-
54         >getAge());
55     }
56 }
57 class UpdateAdvantages implements SplObserver
58 {
59     public function update(SplSubject $person)
60     {
61         // Code métier à exécuter lorsque la classe est notifiée
62         if (18 === $person->getAge()) {
63             echo 'Vous êtes désormais majeur, vous avez de nouveaux avantages, et de nouvelles
64             responsabilités ! <br/>';
65         }
66     }
67 }
68 $observerA = new SayHappyBirthday();
69 $observerB = new UpdateAdvantages();
70 $subject = new Person(16);
71 // On rajoute nos observers à notre sujet
72 $subject->attach($observerA);
73 $subject->attach($observerB);
74
75 $subject->updateAge();
76 $subject->updateAge();
77 $subject->updateAge();
78

```

Méthode Le design pattern Strategy

Le patron de conception Strategy se rapproche beaucoup de l'injection de dépendance.

Sur le principe, les deux design pattern se ressemblent énormément : nous injectons une interface dans une classe afin de découpler notre code.

La différence principale est que, dans le cas de l'injection de dépendance, nous n'avons pas la possibilité de changer l'implémentation de cette interface durant l'exécution de notre script. À l'inverse, dans le cas de Strategy, nous allons pouvoir le faire.

Exemple

Reprenons l'exemple utilisé pour présenter l'injection de dépendances, et modifions-le pour qu'il corresponde au design pattern Strategy.

```

1 <?php
2
3 interface ConnectionInterface
4 {
5     public function getResult(string $query);
6 }
7
8 class PDOConnection implements ConnectionInterface {
9     public function getResult(string $query)
10     {
11         echo 'Results for query with PDOConnection <br/>';
12     }
13 }

```

```

14
15 class MySQLiConnection implements ConnectionInterface {
16     public function getResult(string $query)
17     {
18         echo 'Results for query with MySQLiConnection <br/>';
19     }
20 }
21
22 class Manager
23 {
24     private $connection;
25
26     public function __construct(ConnectionInterface $connection)
27     {
28         $this->connection = $connection;
29     }
30
31     public function setConnection(ConnectionInterface $connection)
32     {
33         $this->connection = $connection;
34     }
35
36     public function query(string $query)
37     {
38         return $this->connection->getResult($query);
39     }
40 }
41
42 $PDO = new PDOConnection();
43 $MySQLi = new MySQLiConnection();
44
45 // Nous initialisons notre manager avec une connexion PDO
46 $manager = new Manager($PDO);
47 $manager->query('foo bar');
48
49 // ...
50
51 // Il nous est possible de modifier la connexion pour désormais utiliser MySQLi
52 $manager->setConnection($MySQLi);
53 $manager->query('foo bar');
54

```

Conseil Et les autres design patterns alors ?

Il est impossible de voir tous les design patterns dans ce cours, il en existe beaucoup trop et cela ne serait pas nécessairement pertinent.

L'important étant de savoir qu'ils existent, et de toujours vérifier si un design pattern ne permet pas de répondre à un besoin lors de la mise en place d'une fonctionnalité spécifique dans une application.

Avec l'expérience, la mise en œuvre et l'utilisation des modèles de conception s'imposera souvent d'elle-même de façon naturelle.

Syntaxe **À retenir**

- Le design pattern Observer permet d'exécuter du code lorsqu'une classe est modifiée, sans pour autant avoir besoin de la modifier, mais simplement en y ajoutant un Observer.
- Le design pattern Strategy est très proche de l'injection de dépendance, la différence majeure entre les deux est que Strategy est à privilégier lorsqu'un changement d'interface peut s'avérer nécessaire durant l'exécution d'un script.
- Il existe bien d'autres design patterns et tous les connaître par cœur n'est pas forcément utile, il faut simplement connaître leur existence afin d'être capable d'identifier leur utilité lorsque l'occasion se présente.

Complément

Une excellente collection de design patterns expliqués en pseudo-code, ainsi qu'implémentés avec différents langages !¹

VII. Exercice : Appliquez la notion

Lorsqu'un nouvel employé rentre dans votre entreprise, plusieurs actions sont systématiquement effectuées et pourraient être automatisées. Vous êtes en charge d'améliorer ce processus.

Vous disposez du code suivant :

```
1 <?php
2
3 class Employee
4 {
5     private $name;
6
7     public function __construct(string $name)
8     {
9         $this->name = $name;
10    }
11
12    public function getName()
13    {
14        return $this->name;
15    }
16 }
17
18 class EmployeeManager
19 {
20     private $employee;
21
22     public function create(Employee $employee)
23     {
24         $this->employee = $employee;
25
26         // code métier qui permettrait ajout de l'employé en BDD
27
28     }
29
30     public function getEmployee()
31     {
32         return $this->employee;
```

¹ <https://refactoring.guru/design-patterns>

```
33     }
34 }
35 $employeeManager = new EmployeeManager();
36 $employee = new Employee('Caroline');
37 $employeeManager->create($employee);
```

Pour réaliser cet exercice, vous pouvez travailler sur l'environnement de travail :



Question

[solution n°3 p.25]

À l'aide du design pattern Observer, mettez en place les événements suivants lorsqu'un nouvel employé est créé.

- Un message indiquera "Bienvenue à notre nouvel employé %nom%" afin de prévenir l'ensemble des équipes d'un nouvel arrivant
- Un second message à destination des services financiers demandera quant à lui de mettre en place le virement nécessaire au paiement du salaire de cet employé

VIII. Auto-évaluation

A. Exercice final

Exercice 1

[solution n°4 p.26]

Exercice

Un Singleton est une classe qui peut être instanciée...

- ☐ Une seule fois
- ☐ Plusieurs fois

Exercice

Les méthodes `__construct()` et `__clone()` d'un Singleton sont nécessairement...

- ☐ Publiques
- ☐ Privées
- ☐ Statiques

Exercice

Dans quel type de propriété l'instance d'un Singleton doit-elle être stockée ?

- ☐ Publique
- ☐ Privée
- ☐ Statique

Exercice

1 <https://repl.it/>

Quelle alternative au Singleton est-il possible d'utiliser pour éviter le fort couplage entre les classes que ce pattern implique ?

- ☐ L'utilisation du pattern Observer
- ☐ L'utilisation de l'injection de dépendances
- ☐ L'utilisation du pattern Factory

Exercice

Indiquez le design pattern utilisé dans le code ci-dessous.

```
1 <?php
2
3 class PatternSample {
4     public static function create($type) {
5         switch ($type) {
6             case 'car': return new Car();
7             case 'bus': return new Bus();
8             case 'bike': return new Bike();
9             default:
10                 throw new Exception('Wrong type passed.');
```

- ☐ Observer
- ☐ Factory
- ☐ Singleton
- ☐ Strategy

Exercice

Comment définir une classe utilisant le pattern Factory ?

- ☐ Il s'agit d'une classe dont le rôle est d'instancier d'autres classes pour une situation donnée
- ☐ Cette classe ne peut être instanciée qu'une seule fois
- ☐ L'utilisation d'une telle classe permet de déporter la logique d'instanciation d'autres objets
- ☐ Le fait d'utiliser une telle stratégie peut être un frein à l'évolution de l'application

Exercice

Quelle interface une classe déclenchant un événement doit-elle implémenter ?

- ☐ SplSubject
- ☐ SplObserver

Exercice

Quelle interface une classe écoutant un événement doit-elle implémenter ?

- ☐ SplSubject
- ☐ SplObserver

Exercice

Au sein de quelle méthode faut-il implémenter le déclenchement de la méthode `update` d'un Observer ?

- ☐ attach
- ☐ notify
- ☐ detach

Exercice

Quelle différence notable existe-t-il entre l'injection de dépendances et l'implémentation du pattern Strategy ?

- ☐ Dans l'injection de dépendances, il est possible de changer l'implémentation de l'interface utilisée en cours d'utilisation, contrairement au pattern Strategy
- ☐ Dans le pattern Strategy, il est possible de changer l'implémentation de l'interface utilisée en cours d'utilisation, contrairement à l'injection de dépendances

B. Exercice : Défi

Pour les besoins d'un site de vente en ligne, vous êtes chargé de mettre en place un système permettant d'informer un client que sa commande a été expédiée.

Le code ci-dessous implémente la classe `Client` et vous fournit une liste de clients que vous devrez contacter.

Un client est toujours contacté selon son moyen de contact favori, représenté par la propriété `$contactWith`.

Selon celui-ci, la méthode `getContactInformation()` permet de retourner son numéro de téléphone ou son adresse e-mail.

```

1
2 /**
3  * Class Client
4  */
5 class Client
6 {
7     public $name;
8     public $contactWith;
9     public $email;
10    public $phoneNumber;
11
12    public function __construct(string $name, string $contactBy, string $email, string
13    $phoneNumber)
14    {
15        $this->name = $name;
16        $this->contactWith = $contactBy;
17        $this->email = $email;
18        $this->phoneNumber = $phoneNumber;
19    }
20
21    public function getContactInformation()
22    {
23        switch ($this->contactWith) {
24            case 'sms':
25                return $this->phoneNumber;
26                break;
27            case 'email':
28                return $this->email;
29                break;
30        }
    }

```

```

31     }
32 }
33
34 $message = "Commande expédiée";
35 $clientsToNotifyToNotify = [];
36
37 $clientsToNotify[] = new Client("Karine", "email", "karine@mail.fr", "01.02.03.04.05.06");
38 $clientsToNotify[] = new Client("Julien", "sms", "julien@mail.fr", "01.02.03.04.05.07");
39 $clientsToNotify[] = new Client("Karim", "sms", "karim@mail.fr", "01.02.03.04.05.08");
40 $clientsToNotify[] = new Client("Justine", "email", "justine@mail.fr", "01.02.03.04.05.09");
41
42
43 foreach ($clientsToNotify as $client)
44 {
45     // Pour chacun des clients, on doit notifier celui-ci selon son moyen de contact favori que
    sa commande a été expédiée.
46 }

```

Pour réaliser cet exercice, vous pouvez travailler sur l'environnement de travail :



Question 1

[solution n°5 p.28]

Chacun des clients de la liste dont vous disposez doit être informé par une notification selon son moyen de contact favori.

À ce jour, un client peut être notifié par e-mail ou par SMS, mais la société envisage, à terme, de prendre contact avec ses clients au moyen de messageries instantanées. Le système que vous allez implémenter devra prendre en compte cette contrainte d'évolutivité.

Selon le script et les informations dont vous disposez, implémentez les classes nécessaires à la gestion de ces notifications, ainsi que le script permettant leur envoi.

Par simplification, on considérera que l'envoi d'une notification consistera à indiquer "%type de notification% de confirmation envoyé à %moyen contact%".

Question 2

[solution n°6 p.30]

Reprenez le code présent dans la solution ci-dessus.

Lorsqu'une notification de type SMS est envoyée, nous souhaitons être informés du fait que celui-ci a bien été remis à son destinataire.

Implémentez la structure nécessaire à la mise en œuvre de ce système. Une méthode `setReceived(bool $isReceived)` permettra de déclencher un changement d'état de cet objet.

Par simplification, on appellera la méthode sous cette forme `$this->setReceived((bool) rand(0, 1));` au sein de la méthode `send` de l'objet représentant la notification SMS.

Solutions des exercices

1 <https://repl.it/>

p.9 Solution n°1

```

1 <?php
2
3 class DatabaseManager
4 {
5     private static $instance;
6
7     private function __construct()
8     {
9     }
10
11     private function __clone()
12     {
13     }
14
15     public static function getInstance(): self
16     {
17         if (!isset(static::$instance)) {
18             static::$instance = new static;
19         }
20
21         return static::$instance;
22     }
23
24     public function connect(): void
25     {
26         // do stuff
27     }
28 }
29
30 $firstInstance = DatabaseManager::getInstance();
31 $secondInstance = DatabaseManager::getInstance();
32
33 if ($firstInstance === $secondInstance) {
34     echo "Il s'agit de la même instance";
35 }
36

```

p.12 Solution n°2

```

1 <?php
2
3 /**
4  * Interface AnimalInterface
5  */
6 interface AnimalInterface
7 {
8     public function getSoundType(): string;
9 }
10
11 /**
12  * Class Dog
13  */
14 class Dog implements AnimalInterface

```

```

15 {
16     public function getSoundType(): string
17     {
18         return 'Aboiement';
19     }
20 }
21
22 /**
23  * Class Cat
24  */
25 class Cat implements AnimalInterface
26 {
27     public function getSoundType(): string
28     {
29         return 'Miaulement';
30     }
31 }
32
33 /**
34  * Class Horse
35  */
36 class Horse implements AnimalInterface
37 {
38     public function getSoundType(): string
39     {
40         return 'Hennissement';
41     }
42 }
43
44 /**
45  * Class AnimalFactory
46  */
47 class AnimalFactory
48 {
49     /**
50      * @param string $animalType
51      * @return AnimalInterface
52      * @throws Exception
53      */
54     public static function load(string $animalType): AnimalInterface
55     {
56         switch ($animalType) {
57             case 'dog':
58                 return new Dog();
59                 break;
60             case 'cat':
61                 return new Cat();
62                 break;
63             case 'horse':
64                 return new Horse();
65                 break;
66             default:
67                 throw new Exception();
68                 break;
69         }
70     }
71 }
72

```



```

73 $animalTypes = ['horse', 'dog', 'mice', 'cat', 'lion'];
74
75 foreach ($animalTypes as $animalType) {
76     try {
77         $animal = AnimalFactory::load($animalType);
78         echo sprintf("%s : %s <br/>", $animalType, $animal->getSoundType());
79     } catch (Exception $e) {
80         echo sprintf("%s : cet animal n'a pas été implémenté dans le système <br/>",
81             $animalType);
82     }
83 }

```

p. 18 Solution n°3

```

1 <?php
2
3 class Employee
4 {
5     private $name;
6
7     public function __construct(string $name)
8     {
9         $this->name = $name;
10    }
11
12    public function getName()
13    {
14        return $this->name;
15    }
16 }
17
18 class EmployeeManager implements SplSubject
19 {
20     private $observers;
21
22     private $employee;
23
24     public function __construct()
25     {
26         $this->observers = new SplObjectStorage();
27     }
28
29     public function attach(SplObserver $observer)
30     {
31         $this->observers->attach($observer);
32     }
33
34     public function detach(SplObserver $observer)
35     {
36         $this->observers->detach($observer);
37     }
38
39     public function notify()
40     {
41         foreach ($this->observers as $observer) {
42             $observer->update($this);

```

```

43     }
44 }
45
46 public function create(Employee $employee)
47 {
48     $this->employee = $employee;
49
50     // code métier qui permettrait l'ajout de l'employé en BDD
51
52     // on notifie les listeners
53     $this->notify();
54 }
55
56 public function getEmployee()
57 {
58     return $this->employee;
59 }
60 }
61
62 class SayWelcome implements SplObserver
63 {
64     public function update(SplSubject $employeeManager)
65     {
66         echo sprintf('Bienvenue à notre nouvel employé %s <br/>', $employeeManager-
67 >getEmployee()->getName());
68     }
69 }
70
71 class AlertFinancialServices implements SplObserver
72 {
73     public function update(SplSubject $employeeManager)
74     {
75         echo sprintf('Un nouvel employé est arrivé dans l\'entreprise, merci de mettre en
76 place le virement pour %s <br/>', $employeeManager->getEmployee()->getName());
77     }
78 }
79
80 $sayWelcome = new SayWelcome();
81 $alertFinancialServices = new AlertFinancialServices();
82 $employeeManager = new EmployeeManager();
83 $employeeManager->attach($sayWelcome);
84 $employeeManager->attach($alertFinancialServices);
85
86 $employee = new Employee('Caroline');
87 $employeeManager->create($employee);

```


Exercice p. 18 Solution n°4

Exercice

Un Singleton est une classe qui peut être instanciée...

☒ Une seule fois

☐ Plusieurs fois

 Un Singleton est une classe qui ne peut être instanciée qu'une seule fois. On ne doit pas pouvoir faire de `new Singleton()`, ni cloner un objet déjà créé.

Exercice

Les méthodes `__construct()` et `__clone()` d'un Singleton sont nécessairement...

- ☐ Publiques
- ☒ Privées
- ☐ Statiques

Exercice

Dans quel type de propriété l'instance d'un Singleton doit-elle être stockée ?

- ☐ Publique
- ☐ Privée
- ☒ Statique

Exercice

Quelle alternative au Singleton est-il possible d'utiliser pour éviter le fort couplage entre les classes que ce pattern implique ?

- ☐ L'utilisation du pattern Observer
- ☒ L'utilisation de l'injection de dépendances
- ☐ L'utilisation du pattern Factory

Exercice

Indiquez le design pattern utilisé dans le code ci-dessous.

```
1 <?php
2
3 class PatternSample {
4     public static function create($type) {
5         switch ($type) {
6             case 'car': return new Car();
7             case 'bus': return new Bus();
8             case 'bike': return new Bike();
9             default:
10                 throw new Exception('Wrong type passed.');
```

- ☐ Observer
- ☒ Factory
- ☐ Singleton
- ☐ Strategy

Exercice

Comment définir une classe utilisant le pattern Factory ?

- ☒ Il s'agit d'une classe dont le rôle est d'instancier d'autres classes pour une situation donnée
- ☐ Cette classe ne peut être instanciée qu'une seule fois
- ☒ L'utilisation d'une telle classe permet de déporter la logique d'instanciation d'autres objets
- ☐ Le fait d'utiliser une telle stratégie peut être un frein à l'évolution de l'application

Exercice

Quelle interface une classe déclenchant un événement doit-elle implémenter ?

- ☒ SplSubject
- ☐ SplObserver

Exercice

Quelle interface une classe écoutant un événement doit-elle implémenter ?

- ☐ SplSubject
- ☒ SplObserver

Exercice

Au sein de quelle méthode faut-il implémenter le déclenchement de la méthode `update` d'un Observer ?

- ☐ attach
- ☒ notify
- ☐ detach

Exercice

Quelle différence notable existe-t-il entre l'injection de dépendances et l'implémentation du pattern Strategy ?

- ☐ Dans l'injection de dépendances, il est possible de changer l'implémentation de l'interface utilisée en cours d'utilisation, contrairement au pattern Strategy
- ☒ Dans le pattern Strategy, il est possible de changer l'implémentation de l'interface utilisée en cours d'utilisation, contrairement à l'injection de dépendances

p. 21 Solution n°5

```

1 <?php
2
3 /**
4  * Class Notification
5  */
6 abstract class Notification
7 {
8     // Tout type de notification doit implémenter une méthode send()
9     protected abstract function send(string $recipient, string $message);
10
11     public function manageNotification($recipient, $message)
12     {
13         $this->send($recipient, $message);
14     }
15 }
```

```

16
17 /**
18  * Class EmailNotification
19  */
20 class EmailNotification extends Notification
21 {
22     protected function send(string $recipient, string $message)
23     {
24         echo sprintf("Email envoyé au %s contenant le message %s <br/>", $recipient,
25 $message);
26     }
27 }
28 /**
29  * Class SMSNotification
30  */
31 class SMSNotification extends Notification
32 {
33     protected function send(string $recipient, string $message)
34     {
35         echo sprintf("SMS envoyé au %s contenant le message %s <br/>", $recipient, $message);
36     }
37 }
38
39 /**
40  * Class NotificationFactory
41  */
42 class NotificationFactory
43 {
44     // La mise en place du modèle Factory permettra de futures évolutions
45     /**
46      * @param string $contactType
47      * @return EmailNotification|SMSNotification
48      */
49     public static function createNotification(string $contactType)
50     {
51         switch ($contactType) {
52             case 'sms':
53                 return new SMSNotification();
54                 break;
55             case 'email':
56             default:
57                 return new EmailNotification();
58         }
59     }
60 }
61
62 /**
63  * Class Client
64  */
65 class Client
66 {
67     public $name;
68     public $contactWith;
69     public $email;
70     public $phoneNumber;
71
72     public function __construct(string $name, string $contactBy, string $email, string
73 $phoneNumber)

```

```

73     {
74         $this->name = $name;
75         $this->contactWith = $contactBy;
76         $this->email = $email;
77         $this->phoneNumber = $phoneNumber;
78     }
79
80     public function getContactInformation()
81     {
82         switch ($this->contactWith) {
83             case 'sms':
84                 return $this->phoneNumber;
85                 break;
86             case 'email':
87                 default:
88                     return $this->email;
89                     break;
90         }
91     }
92 }
93
94 $message = "Commande expédiée";
95 $clientsToNotifyToNotify = [];
96
97 $clientsToNotify[] = new Client("Karine", "email", "karine@mail.fr", "01.02.03.04.05.06");
98 $clientsToNotify[] = new Client("Julien", "sms", "julien@mail.fr", "01.02.03.04.05.07");
99 $clientsToNotify[] = new Client("Karim", "sms", "karim@mail.fr", "01.02.03.04.05.08");
100 $clientsToNotify[] = new Client("Justine", "email", "justine@mail.fr", "01.02.03.04.05.09");
101
102
103 foreach ($clientsToNotify as $client)
104 {
105     $notification = NotificationFactory::createNotification($client->contactWith);
106     $notification->manageNotification($client->getContactInformation(), $message);
107 }

```

p. 21 Solution n°6

```

1 <?php
2
3 /**
4  * Class Notification
5  */
6 abstract class Notification
7 {
8     // Tout type de notification doit implémenter une méthode send()
9     protected abstract function send(string $recipient, string $message);
10
11     public function manageNotification($recipient, $message)
12     {
13         $this->send($recipient, $message);
14     }
15 }
16
17 /**
18  * Class EmailNotification
19  */

```

```

20 class EmailNotification extends Notification
21 {
22     protected function send(string $recipient, string $message)
23     {
24         echo sprintf("Email envoyé au %s contenant le message %s <br/>", $recipient,
25             $message);
26     }
27 }
28 /**
29  * Class SMSNotification
30  */
31 class SMSNotification extends Notification implements SplSubject
32 {
33     private $observers;
34
35     public function __construct()
36     {
37         $this->observers = new SplObjectStorage();
38     }
39
40     public function attach(SplObserver $observer)
41     {
42         $this->observers->attach($observer);
43     }
44
45     public function detach(SplObserver $observer)
46     {
47         $this->observers->detach($observer);
48     }
49
50     public function notify()
51     {
52         foreach ($this->observers as $observer) {
53             $observer->update($this);
54         }
55     }
56
57     protected function send(string $recipient, string $message)
58     {
59         echo sprintf("SMS envoyé au %s contenant le message %s <br/>", $recipient, $message);
60         $this->setReceived((bool) rand(0, 1));
61     }
62
63     public function setReceived(bool $isReceived)
64     {
65         if ($isReceived) {
66             // on ne déclenche l'événement que si le message a bien été reçu
67             $this->notify();
68         }
69     }
70 }
71
72 /**
73  * Class NotificationFactory
74  */
75 class NotificationFactory
76 {

```

```

77  /**
78   * @param string $contactType
79   * @return EmailNotification|SMSNotification
80   */
81  public static function createNotification(string $contactType)
82  {
83      switch ($contactType) {
84          case 'sms':
85              $smsNotification = new SMSNotification();
86              // On attache un type d'événement à notre notification après l'avoir
87  instanciée
88              $smsNotification->attach(new SMSIsReceived());
89              return $smsNotification;
90              break;
91          case 'email':
92              default:
93              return new EmailNotification();
94      }
95  }
96
97  /**
98   * Class SMSIsReceived
99   */
100 class SMSIsReceived implements SplObserver
101 {
102     // le but de cet événement est d'informer l'utilisateur de la bonne remise du message
103     public function update(SplSubject $notification)
104     {
105         echo sprintf('Message remis <br/>');
106     }
107 }
108
109 /**
110 * Class Client
111 */
112 class Client
113 {
114     public $name;
115     public $contactWith;
116     public $email;
117     public $phoneNumber;
118
119     public function __construct(string $name, string $contactBy, string $email, string
120 $phoneNumber)
121     {
122         $this->name = $name;
123         $this->contactWith = $contactBy;
124         $this->email = $email;
125         $this->phoneNumber = $phoneNumber;
126     }
127
128     public function getContactInformation()
129     {
130         switch ($this->contactWith) {
131             case 'sms':
132                 return $this->phoneNumber;
133                 break;

```



```
133         case 'email':
134         default:
135             return $this->email;
136             break;
137     }
138 }
139 }
140
141 $message = "Commande expédiée";
142 $clientsToNotifyToNotify = [];
143
144 $clientsToNotify[] = new Client("Karine", "email", "karine@mail.fr", "01.02.03.04.05.06");
145 $clientsToNotify[] = new Client("Julien", "sms", "julien@mail.fr", "01.02.03.04.05.07");
146 $clientsToNotify[] = new Client("Karim", "sms", "karim@mail.fr", "01.02.03.04.05.08");
147 $clientsToNotify[] = new Client("Justine", "email", "justine@mail.fr", "01.02.03.04.05.09");
148
149
150 foreach ($clientsToNotify as $client)
151 {
152     $notification = NotificationFactory::createNotification($client->contactWith);
153     $notification->manageNotification($client->getContactInformation(), $message);
154 }
```