La Fabrique de Code - 2021

Les Design Patterns

Une courte présentation

- Professionnel depuis 23 ans
- Enseignant
- Auteur publié
- Passionné par l'architecture logicielle
- www.lafabriquedecode.com
- lafabriquedecode@gmail.com

Principes de la conception OO

Isoler les parties susceptibles de varier pour limiter l'impact du changement

Programmer des interfaces (au sens large), pas des implémentations → polymorphisme

Préférer la composition à l'héritage → dyna. vs statique

Limiter le couplage entre les objets, ils doivent pouvoir interagir sans se connaître

L’héritage

```
class Animal {
    public function courir(): void {…}
    public function nager(): void {...}
    public function voler(): void {...}
 }
```

```
class Tigre extends Animal {
    public function voler(): void {...}
}
```

```
class Baleine extends Animal {
    public function courir(): void {...}
    public function voler(): void {...}
}
```

```
class Lombric extends Animal {
  ????
}
```

L’héritage

Attention, il n’est pas question de BANNIR l’héritage, vous allez voir qu’il est utilisé dans de nombreux design patterns

Son usage doit être ré-flé-chi !

La composition

```
class Mere
{
    private Enfant $enfant;

    public function __construct(Enfant $enfant)
    {
        $this->enfant = $enfant;
    }

    public function chatouiller(): void
    {
        $this->enfant->rire();
    }
}
```

Composition

Délégation

La composition - suite

Problème : nous programmons avec des classes concrètes (Enfant)...que se passe-t-il si je crée la classe Nourrisson → **CATASTROPHE** !

Il faut programmer avec des abstractions et non des implémentations ! Ces abstractions sont des **supertypes** (interfaces ou classes abstraites).

La composition - suite

```
interface EnfantInterface
{
    public function rire() : void;
}

class Enfant implements EnfantInterface {}
class Nourrisson implements EnfantInterface {}
```

La composition – suite et fin

```
class Mere
{
    private EnfantInterface $enfant;

    public function __construct(EnfantInterface $enfant)
    {
        $this->enfant = $enfant;
    }

    public function chatouiller(): void
    {
        $this->enfant->rire();
    }
}
```

Les principes SOLID

Des principes de conception orientée objet

Énoncés par Robert C. Martin, Barbara Liskov et Bertrand Meyer (cocorico!)

A toujours garder à l’esprit

Les principes SOLID

S = single responsibility

Ne pas chercher à créer un *God Object*

Les principes SOLID

```
Class GestionnairePersonnes {
    public function creer(array $parametres): void {…}
    public function chercherNom(string $nom) : array {…}
    public function exporterListeComplete(): void {…}
    public function exporterPdf(): void {...}
}
```

Les principes SOLID

```
class GestionnairePersonnes {
    public function creer(array $parametres): void {…}
}

class ExportateurPersonnes {
    public function exporterListeComplete(): void {…}
    public function exporterPdf(): void {...}
}

class RecherchePersonnes {
    public function chercherNom(string $nom): array {…}
}
```

Les principes SOLID

O = ouvert / fermé

Ouvert à l’extension, fermé à la modification

Étendre le comportement sans toucher au code

Solutions : Héritage (abstraction) / Composition

Les principes SOLID

L = principe de substitution de Liskov

Si B dérive du *supertype* A, alors partout où A est utilisé, B pourra l'être sans compromettre le fonctionnement du programme

Les principes SOLID

```
abstract class Animal {
    abstract public function emettreSon(): string;
}

class Chat extends Animal {
    public function emettreSon(): string { return "Miaou";}
}

class Chien extends Animal {
    public function emettreSon(): string { return "Wouf";}
}

class Enfant {
    public function appelleAnimal(Animal $animal): void {
        echo $animal->emettreSon();
    }
}
```

```
$enfant = new Enfant();
$enfant->appelleAnimal(new Chat());
$enfant->appelleAnimal(new Chien());
```

Les deux fonctionnent !

SUPERTYPE = abstrait = Animal
Sous-types = concret = Chien et Chat

Les principes SOLID

I = ségrégation des **interfaces**

Isoler les comportements spécifiques dans des interfaces dédiées

Pas de *Monstro-interface* unique qui force les classes à implémenter des choses inutiles

Les principes SOLID

```
class Dauphin {
    public function nager()...{ }
    public function manger()...{ }
}

class Chien {
    public function courir()...{ }
    public function manger()...{ }
}

class Oiseau {
    public function voler()...{ }
    public function manger()...{ }
}
```

```
abstract class Animal {
     abstract public function manger();
}

interface AnimalNageur {
    public function nager()...;
}

interface AnimalCoureur {
    public function courir()...;
}

interface AnimalVolant {
    public function voler()...;
}
```

```
class Chien
extends Animal
implements AnimalCoureur {
     public function manger()...{ }
     public function courir()...{ }
}
```

Les principes SOLID

Et les oiseaux qui courent ? (autruche)
Et les poissons volants ?
Et les ours, qui courent et nagent ?

Chacun vient choisir son interface !

Les principes SOLID

D = inversion des **dépendances**

Dépendre d’abstractions, pas d’implémentations

Les principes SOLID

```
interface AnimalInterface {
    public function emettreSon() : string;
}

class Enfant {
    public function appelleAnimal(AnimalInterface $animal)
    {
        echo $animal->emettreSon();
    }
}
```

```
class Chat implements AnimalInterface {
    public function emettreSon() : string
    {
        return ‘Miaou’;
    }
}
```

Le composant de haut-niveau Enfant ne dépend pas du composant de bas-niveau Chat mais d’une abstraction, dont dépend également Chat (car il l’implémente)

Les Designs Patterns

Des réponses à des problèmes fréquemment rencontrés par les développeurs

Pas de solution miracle prête à l'emploi, pas de table de la Loi → ne pas chercher à en mettre partout !

Minimiser l'impact des modifications dans le code, favoriser la ré-utilisabilité

Les Designs Patterns

Compilés dans un catalogue édité en 1994 et écrit par le « Gang of Four », devenu LA référence

Un autre ouvrage indispensable : « Design Patterns tête la première » (exemples en Java)

Les Designs Patterns

Au total, 23 design patterns répartis en 3 types :

- Création
- Structure
- Comportement

Les Design Patterns

Patterns de Création

Les Designs Patterns de Création

Gèrent les mécanismes de **création** des objets

Orienté objet : création déléguée à un autre objet

Orienté classe : création déléguée à une sous-classe

Les Designs Patterns de Création

Quelques exemples parmi les plus utilisés :

Factory
Abstract Factory
Builder

Les Designs Patterns de Création

Le système ne doit pas dépendre de la manière dont sont créés les objets

Tout ça est masqué derrière des abstractions

Les Designs Patterns de Structure

Décrivent les moyens de composer des objets pour réaliser de nouvelles fonctionnalités

Composition d’objets = dynamique (exécution)
Composition de classes = statique (compilation)

Les Designs Patterns de Structure

Exemples les plus répandus :

Adaptateur
Décorateur
Façade
Proxy

Les Designs Patterns de Comportement

Ils définissent des modèles :

de classe (utilisent l’héritage)
d’objet (utilisent la composition)
de communication entre eux

Les Designs Patterns de Comportement

Quelques exemples :

Chaîne de Responsabilité
Commande
Observateur
État
Stratégie

Création – Factory

« Fabrique » en français

Orienté **classe** (les sous-classes décident !)

Diminue le couplage entre objets

Création – Factory

Mauvaise pratique

```
class C {
    public function fonc(): void {
        $objet = new D();
    }
}
```

Le couplage entre C et D est maximal ! (le **new**)

Création – Factory

Faisons mieux !

```
class C {
    public function fonc(): void {
        $fabrique = new FabriqueObjetsD();
        $objet = $fabrique->fabriquer();
    }
}
```

Création – Factory

Du côté de FabriqueObjetsD...

```
class FabriqueObjetsD {
    public function fabriquer(): D {
        return new D();
    }
}
```

(dans cet exemple simpliste, une méthode statique suffirait !)

Création – Factory

Si demain D est remplacé par Dbis...

```
class FabriqueObjetsD {
    public function fabriquer(): DBis {
        return new DBis();
    }
}
```

(un seul point de modification, inutile de modifier 250 classes clientes)

Création – Factory

Faisons encore mieux !

```
class C {
    public function fonc(): void {
        $fabrique = new FabriqueObjets('D');
        $objet = $fabrique->fabriquer();
    }
}
```

Création – Factory

Notre FabriqueObjets...

```
class FabriqueObjets {
    public function fabriquer(string $type) {
        if (class_exists($type)) {
            return new $type();
        }
    }
}
```

Création – Factory

Voilà un code qui respecte davantage le O de SOLID

Le code appelant est fermé à la modification

Mieux encore : la fabrique retourne des interfaces

Création – Factory

```
interface ObjetInterface { … }
class D implements ObjetInterface  {...}

class FabriqueObjets {
    public function fabriquer(string $type): ObjetInterface {
        if (class_exists($type)) {
            return new $type();
        }
    }
}
```

Création – Factory

En pratique :

Une interface implémentée par les **fabriques** concrètes

Une interface implémentée par les **produits** concrets

Création – Factory

Autopsie d’une Fabrique

AU TRAVAIL !

Création – Factory

A vous de jouer

Créer **FabriqueDeCarres** et **FabriqueDeTriangles**
Les produits auront une méthode dessiner qui affiche à l'écran un texte.

La méthode de fabrication sera statique !

Création – Factory

A vous de jouer (encore !)

Créer **FabriqueDeConnexions** et **ConnexionMySQL**
La fabrique concrète recevra un tableau de paramètres ainsi que le nom du type de connexion à créer. Connexion avec PDO.
Rappel : DSN constitué de IP et base de données
PDO utilise DSN, *username* et *password*

Création – Factory

Symfony

```
class ConfigCacheFactory implements ConfigCacheFactoryInterface
{
    private $debug;

    public function __construct(bool $debug)
    {
        $this->debug = $debug;
    }

    public function cache($file, $callback)
    {
        $cache = new ConfigCache($file, $this->debug);

         ...

        return $cache;
    }
}
```

Création – Abstract Factory

Même principe...avec des familles de produits

```
interface FabriqueVoitureInterface
{
    public function fabriquerChassis(): PieceInterface;
    public function fabriquerToit(): PieceInterface;
    public function fabriquerPortes(): PieceInterface;
}
```

Création – Patterns Factory

Pour résumer :

Factory : crée un type d’objet
Abstract Factory : plusieurs fabriques créent plusieurs types d’objets fonctionnellement proches (*famille*)

Création – Builder

Appelé MONTEUR en français

Aide à créer des objets complexes

Une classe appelée **directeur** contrôle l'algorithme de construction

Création – Builder

Composants participants :

Le directeur prenant en composition un monteur
Une abstraction pour les monteurs
Des monteurs concrets
Des produits (la cible finale !)
Des parties

Création – Builder

Un exemple, VITE !

Des monteurs de camions et de voitures
Les produits : Camion et Voiture
Les parties : porte, moteur, roue
Un directeur qui orchestre la construction
Une interface pour les monteurs

Création – Builder

A vous de jouer !

Un monteur nommé MonteurDocument
Produit : le document
Parties : entête, corps, pied de page

```
interface DocumentInterface
{
	public function ajouterPartie(string $nom, string $contenu): void;
	public function afficher(): string;
}
```

Création – Builder

Exemple Symfony : FormBuilderInterface

Abstraction pour la construction de formulaires
Produit des formulaires conformes à FormInterface

https://api.symfony.com/master/Symfony/Component/Form/FormBuilderInterface.html

Les Design Patterns

Patterns de Structure

Structure – Adaptateur

Son but : la **conversion** d'interfaces

Analogie récurrente : la prise euro quand vous êtes au Royaume-Uni ou le câble écran

Votre téléphone attend une interface, la prise en propose une autre

Structure – Adaptateur

Adaptateur est composé d'un objet auquel il va fournir une nouvelle interface, différente de celle qu'il proposait

On dit que l'adaptateur *enveloppe* l'objet en composition (*wrapper*)

Structure – Adaptateur

Le client faisait :

```
$objet = new Classe();
$objet->faireAction();
```

Il veut désormais ajouter des classes et unifier :

```
$objet = new Classe();
$objet->faireQuelqueChose();
$autreObjet = new AutreClasse();
$autreObjet->faireQuelqueChose();
$nouvelObjet = new NouvelleClasse();
$nouvelObjet->faireQuelqueChose();
```

Structure – Adaptateur

Notre nouvelle interface :

```
interface MonInterface {
    public function faireQuelqueChose(): void;
}
```

Les classes AutreClasse et NouvelleClasse seront branchées dessus, notre Adaptateur aussi !

Structure – Adaptateur

Notre adaptateur

```
class AdaptateurDeClasse implements MonInterface {
    protected Classe $objet;
    public function __construct(Classe $objet) { …}

    public function faireQuelqueChose(): void {
        $this->objet->faireAction();
    }
}
```

La composition (et la délégation) plutôt que l’héritage

Structure – Adaptateur

En action

```
$objet = new Classe();
$adaptateur = new AdaptateurDeClasse($objet);
$adaptateur->faireQuelqueChose();
```

On ne s’adresse plus directement à une instance de Classe, mais à celle de son adaptateur.

Structure – Adaptateur

Résumé

- On veut utiliser une classe existante, dont l'interface ne correspond pas à celle attendue
- Repose sur la **composition**
- Le client est **découplé** de l'objet adapté
- Méthode de la cible pas implémentée dans l'adapté : l'adaptateur va devoir lancer des exceptions, forçant le client à les gérer

Structure – Adaptateur

A votre tour ! Dans notre appli, nous avions ça :

```
interface LivreInterface
{
	public function tournerPage(): void;
	public function ouvrir(): void;
}

class LivrePapier implements LivreInterface
{
	protected int $page;

	public function tournerPage(): void
	{
		$this->page++;
	}

	public function ouvrir(): void
	{
		$this->page = 1;
	}
}
```

```
class Lecteur
{
	protected LivreInterface $livre;

	public function __construct(LivreInterface $livre)
	{
		$this->livre = $livre;
	}

	public function lire(): void
	{
		$this->livre->ouvrir();
		$this->livre->tournerPage();
	}
}

$lecteur = new Lecteur(new LivrePapier());
$lecteur->lire();
```

Structure – Adaptateur

Nous rachetons un produit qui impose ça, que faire ?

```
interface LiseuseInterface
{
    public function demarrer(): void;
    public function appuyerPageSuivante(): void;
}

class Kobo implements LiseuseInterface
{
    protected int $page;

    public function demarrer(): void
    {
        $this->page = 1;
    }

    public function appuyerPageSuivante(): void
    {
        $this->page++;
    }
}
```

Réponse : créer **AdaptateurLiseuse**, qui permettra à notre nouvelle classe d'être utilisable par l'existant (Lecteur)

Structure – Adaptateur

Symfony – les adaptateurs de cache

AbstractAdapter.php

AdapterInterface.php
ApcuAdapter.php
ArrayAdapter.php
ChainAdapter.php
DoctrineAdapter.php
FilesystemAdapter.php
MemcachedAdapter.php
NullAdapter.php
PdoAdapter.php
PhpArrayAdapter.php
PhpFilesAdapter.php
ProxyAdapter.php
RedisAdapter.php
SimpleCacheAdapter.php
TagAwareAdapterInterface.php
TagAwareAdapter.php
TraceableAdapter.php
TraceableTagAwareAdapter.php

Structure – Décorateur

Le réflexe: dériver une classe pour y ajouter des fonctionnalités → ne pas en abuser ! (sans compter qu'une classe peut être ***finale***)

Décorateur ajoute de nouvelles fonctionnalités en écrivant du code et non en modifiant l'existant (le O de SOLID, vous vous souvenez ?)

Structure – Décorateur

Son but :

Modifier des petites parties d'un objet sans altérer sa structure ni menacer la stabilité du reste de l'application

Ajouter/Retirer des responsabilités à des objets individuellement plutôt qu'à toute une classe.

Structure – Décorateur

Son fonctionnement :

Le décorateur se conforme à l'interface du composant de sorte que sa présence est transparente aux clients du composant

Le décorateur DÉLÈGUE les requêtes au composant et peut effectuer des actions additionnelles

Structure – Décorateur

Comme Adaptateur, c’est une **enveloppe**, mais il ne modifie pas l’interface, il lui ajoute des comportements.

Composition = Dynamique (*runtime*)
Héritage = Statique (compilation)

Structure – Décorateur

A privilégier quand le nombre de sous-classes serait tellement élevé qu'il conduirait à une explosion combinatoire de celles-ci.

Ex : une classe Voiture, VoitureAToitOuvrant, VoiturePeintureMetal, VoitureAToitOuvrantEtPeintureMetal etc.

Structure – Décorateur

Abus des décorateurs = sous-classes nombreuses

Difficulté de maintenance

Appels imbriqués difficilement lisibles

Structure – Décorateur

Rentrons dans le vif du sujet avec des décorateurs de véhicules.

Créez un décorateur Climatisation (prix 1000€) et les véhicules suivants :

- Clim seule
- Clim et toit ouvrant
- Clim, peinture rouge et toit ouvrant

Structure – Décorateur

A votre tour de décorer !

```
class Burger implements BurgerInterface
{
    public function getDescription(): string {
        return 'Steak, Cheddar';
    }

    public function getPrix(): float {...}
}

class BurgerVeggie implements BurgerInterface
{
    public function getDescription(): string {
        return 'Steak de tofu, fauxmage';
    }

    public function getPrix(): float {...}
}
```

Je veux des ingrédients supplémentaires :
- bacon
- cornichon

Les décorateurs **SupplementBacon** et **SupplementCornichon** vont venir décorer des objets implémentant le **supertype BurgerInterface**

ATTENTION : pas de bacon dans mon burger veggie, merci !

Structure – Décorateur

A votre tour de décorer (encore !)

Voici le **supertype** (l'abstraction, l'interface au sens large) de la classe concrète **FabriqueDeSupplement** que vous allez créer :

```
interface FabriqueDeSupplementInterface
{
    public function fabriquer(string $type): BurgerInterface;
}
```

Cette fabrique « hybride » de décorateurs, pas vraiment une **Factory** puisqu'elle ne crée pas un seul type d'objet et pas vraiment une **Abstract Factory** non plus puisqu'elle n'a pas de notion de famille d'objets, va retourner des décorateurs du type demandé (bacon ou cornichon).

Elle prendra en composition toute classe se conformant à l'interface **BurgerInterface**.
Elle appliquera le supplément demandé et renverra le burger ainsi décoré.

Modifiez le code pour faire en sorte que la fabrique soit utilisée en lieu et place des décorateurs directement.

Faites en sorte de gérer le cas où le supplément demandé n'existe pas en lançant une exception du type de votre choix.

Structure – Décorateur

Symfony

```
$kernel = new AppKernel('prod', false);
$kernel = new AppCache($kernel);
```

AppCache vient décorer AppKernel pour lui ajouter des fonctionnalités de gestion du cache

Structure – Façade

Masque la complexité d’un sous-système

Propose une interface simplifiée au client, qui ignore tout de la complexité sous-jacente

Façade délègue les appels aux composants du sous-système

Structure – Façade

La classe Façade est un facilitateur qui découple le sous-système du ou des clients

Le sous-système reste toutefois utilisable directement, la façade n’est pas un point d’entrée unique !

Structure – Façade

Voyons tout ça en action !

Structure – Façade

A vous de faire !

```
interface LecteurInterface
{
    public function mettreSousTension(): void;
    public function lire(): void;
}

interface ProjecteurInterface
{
    public function mettreSousTension(): void;
    public function modePleinEcran(): void;
}

interface AmplificateurInterface {
    public function mettreSousTension(): void;
    public function activerSonSurround(): void;
    public function reglerVolume(int $volume): void;
}
```

```
class LecteurBluRay implements LecteurInterface
{
    protected string $film;

    public function nomduFilm(string $film) : void
    {
        $this->film = $film;
    }
}
```

1) Complétez la classe **LecteurBluRay**
2) Créez la façade **HomeCinemaFacade** et sa méthode *regarderFilm (string $film, int $volume)*

Structure – Proxy

Se substitue à un objet (le *sujet réel*) pour éventuellement en contrôler l'accès

Délègue à cet autre objet

On utilise aussi Proxy pour faire du *caching* de requêtes (SQL ou HTTP)

Structure – Proxy

Passons à la pratique !

Structure – Proxy

Maintenant c’est à vous !

```
interface UtilisateurInterface
{
    public function afficherDossierMedical(): ?array;
    public function estSuperAdmin(): bool;
}

class Utilisateur implements UtilisateurInterface
{
        protected string $login;
        protected string $motdepasse;

        public function __construct(string $login, string $motdepasse) {
                $this->login = $login;
                $this->motdepasse = $motdepasse;
        }

        public function afficherDossierMedical(): ?array {
                return null;
        }

        public function estSuperAdmin(): bool {
             return false;
        }
}
```

Créer **ProxyUtilisateur** qui n’affiche le Dossier médical que si l’utilisateur est super admin

Créez une classe **SuperUtilisateur** dont la méthode estSuperAdmin renverra true et afficherDossierMedical un dossier médical

Faisons simple !

Structure – Proxy

Symfony

```
namespace Symfony\Component\HttpFoundation\Session\Storage\Proxy;

class SessionHandlerProxy extends AbstractProxy implements \SessionHandlerInterface, \SessionUpdateTimestampHandlerInterface
{
    protected $handler;

    public function __construct(\SessionHandlerInterface $handler)
    {
        $this->handler = $handler;
    }

    public function open($savePath, $sessionName)
    {
        return (bool) $this->handler->open($savePath, $sessionName);
    }
}
```

Les Design Patterns

Patterns de Comportement

Comportement – Commande

Encapsulation de requêtes

Découpler l’objet qui demande une requête de l’objet qui l’exécute

Comportement – Commande

Les composants :

- le **client** : il créé l'objet commande
- un **invocateur** qui exécute la commande
- la **commande** et son abstraction qui ont une méthode executer()
- le **récepteur** : en bout de chaîne, c'est lui qui réalise l'action

Comportement – Commande

Voyons en détail comment tout cela fonctionne !

Comportement – Commande

Exercice :

- le **récepteur** : Lampe (on l'allume et on l'éteint...deux commandes, donc !)
- l'**invocateur** : Telecommande (on lui injecte des commandes) et son setter *changerCommande*

```
interface MettableSousTensionInterface
{
        public function allumer(): void;
        public function eteindre(): void;
}
```

Comportement – Commande

Symfony

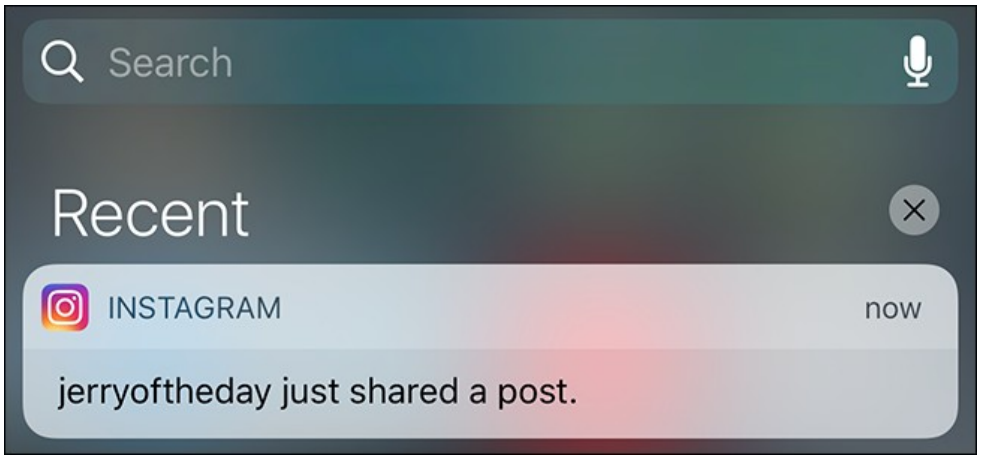
```
namespace Symfony\Component\Console\Command;

class Command       (Cette classe est concrète)
{
    protected function execute(InputInterface $input, OutputInterface $output)
    {
        throw new LogicException('You must override the execute() method in the concrete command class.');
    }
}

class CacheClearCommand extends Command
{
    public function __construct(CacheClearerInterface $cacheClearer, Filesystem $filesystem = null)
    {
    }
}
```

Comportement – Observateur

Des **observateurs** qui « écoutent » un sujet qui les informe de tout changement de son état...ça ne vous rappelle rien ?



Vous êtes l'observateur du sujet **jerryoftheday**. Son état change (il a publié un message), vous en êtes notifiés !

Comportement – Observateur

Relation 1 à N entre le sujet et ses observateurs

Le sujet doit ajouter/supprimer des observateurs et les notifier de tout changement...

Comportement – Observateur

L’observateur doit implémenter une méthode de mise à jour (abstraction)

Aucune modification du sujet lors de l’ajout d’un observateur, il suffit juste qu’il implémente l’interface → le O de SOLID, encore !

Comportement – Observateur

La notification des observateurs ne doit pas se faire dans un ordre précis, sinon un couplage inutile est créé !

Le sujet peut **pousser** les données ou laisser les observateurs les **tirer** (conseillé)

Comportement – Observateur

Pousser : les observateurs ont l'intégralité des modifications, que cela leur serve ou non.

Ce modèle sous-entend que les sujets savent ce dont ont besoin leurs observateurs...

Tirer : les observateurs sont tenus d'aller chercher ce qui a effectivement changé

Comportement – Observateur

PHP propose des interfaces pour gérer ce pattern :

```
interface SplObserver {
    public update ( SplSubject $subject ) : void;
}

interface SplSubject {
    public attach ( SplObserver $observer ) : void;
    public detach ( SplObserver $observer ) : void;
    public notify ( ) : void;
}
```

Push ou Pull ?

Comportement – Observateur

A présent, observons !

Comportement – Observateur

C’est à vous de Parier !

(faire avec la SPL pour les plus vaillant(e)s!)

Comportement – Observateur

Symfony

Ce design pattern est implémenté dans le système d'événements du framework :

EventDispatcher = **Sujet**
EventListeners et EventSubscribers = **Observateurs**

Comportement – Stratégie

Définit une famille d’algorithmes qu’on encapsule et qui sont interchangeables (L de SOLID)

Toujours le même bon vieux principe : une abstraction et des réalisations qui l’implémentent → Couplage réduit

Comportement – Stratégie

Un objet appelé **contexte** possède en composition l'objet stratégie auquel il délègue le travail (cette dépendance lui est injectée)

VITE, un exemple !

Comportement – Stratégie

Un objet appelé **contexte** possède en composition l'objet stratégie auquel il délègue le travail (cette dépendance lui est injectée)

VITE, un exemple !

Comportement – Stratégie

A vos stratégies !

```
interface ValidateurInterface
{
    public function valider(UtilisateurInterface $utilisateur): bool;
}

class AuMoinsUnChiffre implements ValidateurInterface {
        A FAIRE !
}

class AuMoinsDixCaracteres implements ValidateurInterface {
        A FAIRE !
}

class PasDespace implements ValidateurInterface {
    A FAIRE !
}
```

```
class Contexte {
    A FAIRE !
}

interface UtilisateurInterface {
    public function donnerMotDePasse(): string;
}

class Utilisateur implements UtilisateurInterface {
    protected string $motDePasse;

    public function __construct(string $motDePasse)
    {
        $this->motDePasse = $motDePasse;
    }
    A COMPLETER !
}
```

Comportement – Stratégie

Symfony

Le système de validation est basé sur ce design pattern :

Un contexte d’exécution
Des validateurs
Des listes de violations des contraintes

Comportement – État

Permet à un objet de modifier son comportement quand son état change

Les états implémentant la même interface, ils deviennent interchangeables (coucou Liskov !)

Il comprend un contexte, comme Stratégie !

Comportement – État

Ce contexte délègue lui aussi à l'objet qu'il a en composition

Dans État, le client ne sait rien des objets états alors que dans Stratégie, c'est lui qui les injecte dans le contexte

Comportement – État

Le contexte garde la trace de l'état courant

Potentiellement de nombreuses classes État
mais c'est le prix de la souplesse

Dans notre exemple, les classes État donnent les transitions, mais on peut le faire dans le contexte (si elles sont statiques)

Comportement – État

1 état par classe = nous isolons les futurs changements

Le client ne parle qu’avec le contexte, dont l’état varie au fur et à mesure des transitions

Comportement – État

Maestro, exemple !

Comportement – État

A vous de jouer maintenant...

Comportement – Template Method

Patron de méthode en bon français

Un pattern très simple (promis !)

Ce patron (*template*) de méthode se trouve dans une classe abstraite

Comportement – Template Method

Le patron définit les étapes (le squelette) d'un algorithme et laisse les sous-classes les implémenter (ou pas)

On appelle ça le « principe de Hollywood ». La classe mère appelle les classes filles dans son squelette.

Comportement – Template Method

C’est vous les patrons désormais, au boulot !

```
abstract class PersonneAbstract
{
    public function routine(): void
    {
        if ($this->aUnJob()) {
          $this->partirTravailler();
        }

        if ($this->aUnJob()) {
            $this->rentrerDuTravail();
        }
    }

    abstract protected function seLever(): void;
    abstract protected function partirTravailler(): void;
    abstract protected function dejeuner(): void;
    abstract protected function rentrerDuTravail(): void;
    abstract protected function diner(): void;
    abstract protected function dormir(): void;
}
```

Placez les méthodes manquantes à l’endroit qui convient dans le patron routine()

Créez deux classes concrètes Chomeur et Patron et faites fonctionner le tout !

Comportement – Template Method

Symfony

```
namespace Symfony\Component\Config\Definition;

abstract class BaseNode implements NodeInterface {
        final public function finalize($value)
        {
                $this->doValidateType($value);
                $value = $this->finalizeValue($value);
        }
}

abstract protected function normalizeValue($value);
abstract protected function mergeValues($leftSide, $rightSide);
abstract protected function finalizeValue($value);

protected function allowPlaceholders(): bool {
    return true;
}
```

Comportement – Itérateur

Fournit un moyen d'accéder séquentiellement à un objet de type agrégat sans révéler sa représentation sous-jacente (tableau par ex.)

De nombreux itérateurs sont intégrés dans PHP: http://php.net/manual/fr/spl.iterators.php

Comportement – Itérateur

PHP a deux interfaces qu'il faut connaître :

IteratorAggregate : la classe qui détient l'agrégat en composition l'implémente

Iterator : implémentée par l'itérateur

Comportement – Itérateur

Attention, un itérateur a pour but de parcourir, pas d’ordonner !

Comportement – Itérateur

Entrons dans les détails

Comportement – Itérateur

Prêts ? Itérez !

Vous compléterez l'itérateur existant puis créerez un itérateur **IterateurARebours** qui parcourt la collection en sens inverse

Comportement – Itérateur

Symfony fait un usage intensif des itérateurs de la SPL et a implémenté les siens propres

```
class OrderedHashMap implements \ArrayAccess, \IteratorAggregate, \Countable
{
    public function getIterator()
    {
        return new OrderedHashMapIterator($this->elements, $this->orderedKeys, $this->managedCursors);
    }
}

class OrderedHashMapIterator implements \Iterator
{
}
```

Comportement – Chaîne de Resp.

Une requête, plusieurs récepteurs chaînés susceptibles de la traiter

La requête parcourt la chaîne jusqu'à ce qu'elle soit prise en charge par 1 ou N récepteurs

L'objet émetteur de la requête ne sait pas qui va la traiter → couplage réduit

Comportement – Chaîne de Resp.

Les acteurs :

Une **abstraction** pour les gestionnaires, des **gestionnaires concrets** qui savent comment accéder à leur successeur dans la chaîne

Un client qui propose la requête à un gestionnaire concret pour traitement

Comportement – Chaîne de Resp.

Passons aux exemples !

Le Design Pattern MVC

Pas un pattern GoF

Inventé en 1978 !!!

3 composants, comme son nom l’indique

Le Design Pattern MVC

La vue : ce que l’utilisateur a sous les yeux

Le contrôleur : la logique, c’est là où résident les algorithmes

Le modèle : l’interface avec les données

Le Design Pattern MVC

En théorie :
M ne se sert ni de V, ni de C
C agit sur M et V
V interroge C

Dans la plupart des frameworks MVC, les deux premiers sont vrais