Les Design Patterns
(Descriptions de 23 modèles de conception)

N° de la lecture individuelle :	1
Semestre	3
Étudiant	DAVID Guillaume, 803_1F
Sujet	Design Patterns

Choix du sujet

J'ai décidé de réaliser cette lecture individuelle axée sur les modèles de conception logicielle ou autrement dit les « Design Patterns » pour diverses raisons.

Ma première motivation est mon souhait de pouvoir améliorer l'organisation de mon code ainsi que l'organisation et la construction de l'architecture logicielle d'un projet. Je pense que je pourrais apprendre davantage et améliorer ma manière de créer des systèmes étudiées et évolutifs en explorant les astuces et les méthodes les plus efficaces liées aux « Design Patterns ».

Par ailleurs, je souhaite prendre le temps de comprendre ce domaine. J'ai déjà quelques notions grâce à mon expérience passée mais je souhaite avoir au moins connaissance des principes et solutions étudiées à des problématiques connues.

Enfin, je prévois d'appliquer ce que je vais apprendre à des projets concrets. Deux projets sont en vue : l'un en PHP et l'autre en Dart. En travaillant avec ces langages, je pourrais mettre en pratique ce que j'ai appris, non seulement pour renforcer ma compréhension théorique, mais aussi pour créer des applications réelles qui intègrent ces idées de conception.

Table des matières

Choix du sujet 1
Support théorique 3
Titre 3
Auteurs 3
Organisation du résumé 3
Introduction 4
Qu’est-ce qu’un « Design Pattern » ? 4
Les principes SOLID 4
S - Single Responsibility Principle (Principe de Responsabilité Unique) 4
O - Open-Closed Principle (Principe Ouvert/Fermé) 5
L - Liskov Substitution Principle (Principe de Substitution de Liskov) 5
I - Interface Segregation Principle (Principe de Ségrégation des Interfaces) : 6
D - Dependency Inversion Principle (Principe d'Inversion des Dépendances) 6
Les « Design Patterns » 6
Les 23 designs patterns du « GoF » 7
Les design patterns de construction 8
Problématique 8
Singleton [facile] 9
Builder [utile] 10
Factory Method [difficile / utile] 11
Les design patterns de structuration 13
Adapter [Facile] 13
Composite [Utile] 15
Flyweight [Difficile] 17
Les design patterns de comportement 18
Memento [Facile] 19
Proxy [Utile] 20
Command [Difficile] 22
Conclusion générale 24
Conclusion personnelle 24

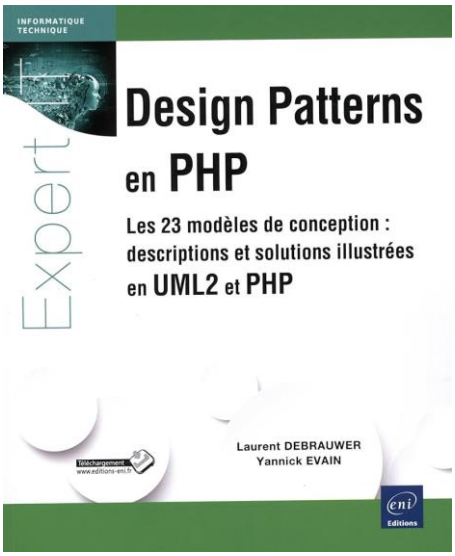
Support théorique

La recherche se base principalement sur le livre présenté ci-dessous. Des apports, de l'aide à la construction des exemples, et des compréhensions ont également réalisés avec ChatGPT.

Titre

Design Patterns en PHP, Les 23 modèles de conception : descriptions et solutions illustrées en UML2 et PHP (2e édition)

Éditions ENI, 2021, Réf. ENI : EI2PHDES | ISBN : 9782409030673



Auteurs

Laurent DEBRAUWER : Docteur en informatique, auteur et dirigeant des sociétés META-AGENT Software et Semantica, spécialisé en linguistique et sémantique. Enseigne les Design Patterns à l'université du Luxembourg.

Yannick EVAIN : Élève ingénieur en dernière année à Télécom Lille, avec une expérience de 10 ans à la SNCF dans la conception de systèmes électroniques numériques et informatiques pour le matériel roulant.

Sébastien FERRANDEZ : Ingénieur diplômé du Conservatoire National des Arts et Métiers d'Aix-en-Provence, consultant et formateur indépendant, spécialisé en développement web, bases de données (relationnelles et NoSQL) et enseignement des Design Patterns et du langage PHP.

Organisation du résumé

Pour ce résumé, étant donné la diversité des design patterns et leur utilité variable, j'ai opté pour une approche méthodique. Je vais d'abord expliquer en détail les différentes catégories de design patterns, puis je vais présenter un exemple facile, un exemple utile, et un exemple que je juge particulièrement complexe de chaque catégorie. Les autres design patterns seront brièvement mentionnés sans entrer dans les détails.

De plus, je vais aborder la section consacrée aux principes SOLID.

Introduction

Cet ouvrage simplifie et présente de manière efficace les 23 modèles de conception issus du célèbre livre « Design Patterns - Elements of Reusable Object-Oriented Software » du « Gang of Four » ou « GoF ».

Qu'est-ce qu'un « Design Pattern » ?

Un design pattern, ou modèle de conception, représente une solution à des problèmes de conception récurrents en programmation orientée objet (POO).

La vision « GoF » structure les design patterns en trois grandes catégories qui sont :

1. Les **design patterns de création** qui visent à simplifier la création d'objets en encapsulant les mécanismes d'instanciation des classes concrètes. Cela permet de minimiser les modifications nécessaires en cas de changement des classes à instancier.

2. Les **design patterns structurels** qui ont pour objectif d'abstraire l'interface d'un objet ou d'un ensemble d'objets de son implémentation, ou encore d'abstraire l'interface des relations d'héritage ou de composition dans un ensemble d'objets.

3. Les **design patterns de comportement** qui offrent des solutions pour organiser les interactions entre objets, distribuer les traitements et les algorithmes, ainsi que pour structurer les données.

Les principes SOLID

Les principes SOLID sont un ensemble de **5 principes fondamentaux** en programmation orientée objet (POO) qui visent à guider les développeurs vers la création de code robuste, flexible et facile à maintenir. Chacune de ces lettres correspond à un principe spécifique :

S - Single Responsibility Principle (Principe de Responsabilité Unique)

Ce principe stipule qu'une classe ne devrait avoir qu'une seule raison d'exister. En d'autres termes, une classe doit avoir une seule responsabilité ou fonction. Cela favorise la cohésion et la modularité du code en évitant que les classes ne deviennent trop complexes.

La classe suivante a été réécrite afin de satisfaire une seule responsabilité.

AVANT	APRES

```php
class Logger
{
    protected string $type;


    public function __construct(string $type)
    {
        if (!in_array(strtolower($type), ['fichier', 'bdd'])) {
            throw new InvalidArgumentException('Type non pris en
charge');
        }

        $this->type = $type;
    }

    public function logDansFichier(string $nomFichier, string
$information): void
    {
        if ($this->type == 'fichier' && is_writable($nomFichier)) {
            // ...
        }
    }

    public function logDansBaseDonneesSQL(string $chaineConnexion,
string $information): void
    {
        if ($this->type == 'bdd' && !empty($chaineConnexion)) {
            // ...
        }
    }
}
```

```php
class LoggerFichier
{
    protected string $nomFichier;

    public function __construct(string $nomFichier)
    {
        if (!is_writable($nomFichier)) {
            throw new InvalidArgumentException('Le fichier est
inaccessible en écriture');
        }

        $this->nomFichier = $nomFichier;
    }

    public function ecrire(string $information): void
    {
        if (empty($information)) {
            throw new InvalidArgumentException('Aucune information
à logger');
        }

        file_put_contents($this->nomFichier, $information .
PHP_EOL, FILE_APPEND | LOCK_EX);
    }
}

class LoggerBaseDonneesSQL
{
    protected string $chaineConnexion;

    public function __construct(string $chaineConnexion)
    {
        if (empty($chaineConnexion)) {
            throw new InvalidArgumentException('La chaîne de
connexion est vide');
        }

        $this->chaineConnexion = $chaineConnexion;
    }

    public function ecrire(string $information): void
    {
        // INSERT INTO log(date, information) values (now(),
$information) etc.
    }
}
```

```php
interface LoggerInterface
{
    public function ecrire(string $information): void;
}
```

O - Open-Closed Principle (Principe Ouvert/Fermé)

Ce principe énonce que les entités logicielles, telles que les classes, les modules ou les fonctions, doivent être ouvertes à l'extension mais fermées à la modification. En d'autres termes, vous devriez pouvoir ajouter de nouvelles fonctionnalités sans avoir à modifier le code existant. Les interfaces jouent un rôle important.

Si l'on reprend l'exemple, l'implémentation d'une interface permet de d'éviter les modifications.

AVANT

```php
if ($cible == 'bdd') {
    $logger = new Logger($cible);
    $logger->logDansBaseDonneesSQL('chainedeconnexion', 'debut
écriture');
} else {
    $logger = new Logger($cible);
    $logger->logDansFichier('/tmp/test', 'debut écriture');
}
```

APRES

```php
if ($cible == 'bdd') {
    $logger = new LoggerBaseDonneesSQL('chainedeconnexion');
} else {
    $logger = new LoggerFichier('/tmp/test');
}
$logger->ecrire('debut écriture');
```

L - Liskov Substitution Principle (Principe de Substitution de Liskov)

Ce principe stipule que les objets d'une sous-classe doivent pouvoir être utilisés de manière interchangeable avec les objets de leur classe de base (superclasse) sans perturber le comportement correct du programme. Il garantit que les sous-classes respectent les contrats définis par les superclasses.

```php
function ecrireInformation(LoggerInterface $logger, string
$information): void
{
    $logger->ecrire($information);
}
```

I - Interface Segregation Principle (Principe de Ségrégation des Interfaces) :

Ce principe encourage la définition d'interfaces spécifiques aux besoins des clients plutôt que de créer des interfaces générales qui regroupent de nombreuses méthodes. Il vise à éviter que les classes clients soient forcées d'implémenter des méthodes dont elles n'ont pas besoin.

ACTION	IMPACT

`interface AnimalInterface`
`{`
`  public function voler(): void;`
 	
`  public function nager(): void;`
`}`
	`class Baleine implements AnimalInterface`
	`{`
	`  public function voler(): void`
	`  {`
	`  throw new \Exception('Je ne sais pas voler');`
	`  }`
	
	`  public function nager(): void`
	`  {`
	`  echo 'Je nage';`
	`  }`
	`}`

`interface AnimalNageurInterface`
`{`
`  public function nager(): void;`
`}`
`interface AnimalVolantInterface`
`{`
`  public function voler(): void;`
`}`	

D - Dependency Inversion Principle (Principe d'Inversion des Dépendances)

Ce principe encourage l'inversion de la dépendance entre les modules. goCela favorise la flexibilité et la réutilisation du code en réduisant les couplages entre les composants.

RESPECT	NE RESPECTE PAS

`class GestionnaireEvenement`
`{`
`  public function enregistrerEvenement(LoggerInterface $logger,`
`string $evenement): void`
`  {`
`  $logger->ecrire($evenement);`
`  }`
`}`
	`class GestionnaireEvenement {`
	`  public function enregistrerEvenement(FileLogger $logger, string $eveneme`
	`  $logger->ecrire($evenement);`
	`  }`
	`}`
	
	`class FileLogger {`
	`  public function ecrire(string $message): void {`
	`  // Enregistre le message dans un fichier`
	`  }`
	`}`

En respectant ces principes, les développeurs sont en mesure de créer des systèmes logiciels plus modulaires, extensibles et faciles à maintenir. Les principes SOLID sont largement utilisés en développement logiciel pour promouvoir une meilleure conception orientée objet et une gestion plus efficace de la complexité du code.

Les « Design Patterns »

Les design patterns sont des solutions éprouvées pour résoudre des problèmes courants en programmation orientée objet, basées sur de bonnes pratiques. Ils ne sont pas stricts et peuvent être adaptés ou créés selon les besoins. Bien que précieux pour assurer la qualité du code, ils ne sont pas indispensables. Il est important de ne pas les utiliser de manière excessive, car cela

peut être contre-productif. **Les design patterns sont un outil utile pour les concepteurs et les développeurs, permettant de nommer et de partager des solutions éprouvées depuis des décennies.**

Les 23 designs patterns du « GoF »

Designs Pattern de conception	
Abstract Factory	A pour objectif la création d'objets regroupés en familles sans devoir connaître les classes concrètes destinées à la création de ces objets
Builder	Permet de séparer la construction d'objets complexes de leur implémentation de sorte qu'un client puisse créer ces objets complexes avec des implémentations différentes.
Factory Method	A pour but d'introduire une méthode abstraite de création d'un objet en déléguant aux sous-classes concrètes la création effective.
Prototype	Permet la création de nouveaux objets par duplication d'objets existants appelés prototypes qui disposent de la capacité de clonage.
Singleton	Permet de s'assurer qu'une classe ne possède qu'une seule instance en mémoire et de fournir une méthode unique retournant cette instance.
Designs Pattern de structuration	
Adapter	A pour but de convertir l'interface d'une classe existante en l'interface attendue par des clients également existants afin qu'ils puissent collaborer.
Bridge	A pour but de séparer les aspects conceptuels d'une hiérarchie de classes de leur implémentation.
Composite	Offre un cadre de conception d'une composition d'objets dont la profondeur de composition est variable, la conception étant basée sur une structure arborescente.
Decorator	Permet d'ajouter dynamiquement des fonctionnalités supplémentaires à un objet.
Facade	A pour but de regrouper les interfaces d'un ensemble d'objets en une interface unifiée rendant cet ensemble plus simple à utiliser.
Flyweight	Facilite le partage d'un ensemble important d'objets dont la granularité est fine.
Designs Pattern d'organisation	
Proxy	Construit un objet qui se substitue à un autre objet et contrôle son accès.
Chain of Responsability	Crée une chaîne d'objets telle que si un objet de la chaîne ne peut pas répondre à une requête, il puisse la transmettre à ses successeurs jusqu'à ce que l'un d'entre eux y réponde.
Command	A pour objectif de transformer une requête en un objet, facilitant des opérations comme l'annulation, la mise en file des requêtes et leur suivi.
Interpreter	Fournit un cadre pour donner une représentation par objets de la grammaire d'un langage afin d'évaluer, en les interprétant, des expressions écrites dans ce langage.
Iterator	Fournit un accès séquentiel à une collection d'objets sans que les clients se préoccupent de l'implémentation de cette collection.

Mediator	Construit un objet dont la vocation est la gestion et le contrôle des interactions au sein d'un ensemble d'objets sans que ses éléments se connaissent mutuellement.
Memento	Sauvegarde et restaure l'état d'un objet
Observer	Construit une dépendance entre un sujet et des observateurs de façon à ce que chaque modification du sujet soit notifiée aux observateurs afin qu'ils puissent mettre à jour leur état.
State	Permet à un objet d'adapter son comportement en fonction de son état interne.
Strategy	Adapte le comportement et les algorithmes d'un objet en fonction d'un besoin sans changer les interactions avec les clients de cet objet.
Template Method	Permet de reporter dans des sous-classes certaines étapes de l'une des opérations d'un objet, ces étapes étant alors décrites dans les sous-classes.
Visitor	Construit une opération à réaliser sur les éléments d'un ensemble d'objets. De nouvelles opérations peuvent ainsi être ajoutées sans modifier les classes de ces objets.

Les design patterns de construction

« Les **design patterns de construction** ont pour vocation d'abstraire les mécanismes permettant de créer des objets. Un système utilisant ces design patterns devient indépendant de la façon dont les objets sont créés et, en particulier, des mécanismes d'instanciation des classes concrètes »

Cette catégorie comprend cinq modèles : Abstract Factory, Builder, Factory Method, Prototype et Singleton.

Problématique

Le problème central ici réside dans la difficulté à paramétrer le processus de création d'objets dans la programmation orientée objet, ce qui entraîne une dépendance excessive aux détails de l'implémentation. Cela se traduit par l'utilisation d'instructions conditionnelles dans le code client pour choisir la classe à instancier, ce qui devient complexe et nécessite des modifications importantes chaque fois qu'il y a un changement dans la hiérarchie des classes à créer. Cette problématique est encore plus critique lorsqu'il s'agit de construire des objets composés à partir de composants instanciés à partir de classes différentes.

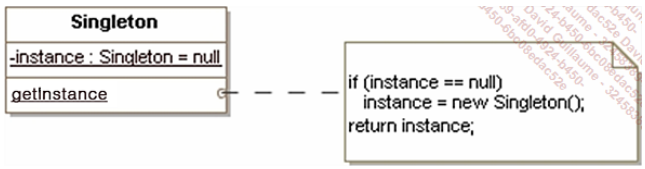
Illustration de la problématique

```php
class ConstructeurDocument
{
    protected Document $resultat;
    public function construitDoc(string $typeDoc): Document
    {
        if ($typeDoc == 'PDF') {
            $resultat = new DocumentPDF();
        } elseif ($typeDoc == 'RTF') {
            $resultat = new DocumentRTF();
        } else {
            $resultat = new DocumentHTML();
        }
        // suite de la méthode
    }
}
```

Singleton [facile]

Le design pattern Singleton vise à garantir qu'une classe a une seule instance en mémoire et offre une méthode de classe permettant d'obtenir cette unique instance.

Diagramme de classe



Example en code PHP

Dans cet exemple, la classe **Configuration** est un Singleton qui stocke les données de configuration. Il est possible d'ajouter ou de récupérer des valeurs de configuration à l'aide des méthodes **set** et **get**. L'appel à **getInstance** garantit qu'une seule instance de la classe **Configuration** est créée et utilisée pour accéder aux données de configuration.

```php
<?php

class Configuration
{
    private static $instance;
    private $configData = [];

    private function __construct()
    {
        $this->configData['database_host'] = 'localhost';
        $this->configData['database_user'] = 'username';
        $this->configData['database_pass'] = 'password';
    }

    public static function getInstance()
    {
        if (self::$instance === null) {
            self::$instance = new Configuration();
        }

        return self::$instance;
    }

    public function get($key)
    {
        if (isset($this->configData[$key])) {
            return $this->configData[$key];
        }
        return null;
    }
```

```
    public function set($key, $value)
    {
        $this->configData[$key] = $value;
    }
}
```

Utilisations

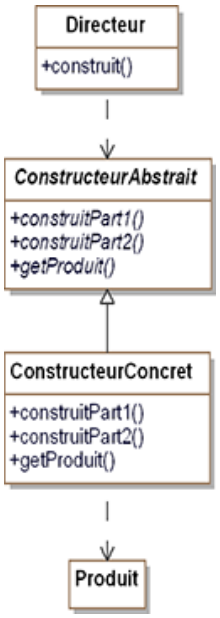
Le design pattern Singleton est appliqué dans les situations suivantes :

1. Lorsqu'il est impératif d'avoir une seule et unique instance d'une classe en mémoire à un moment donné.
2. Quand l'accès à cette instance doit être strictement contrôlé et limité aux méthodes de classe de la classe Singleton.

Builder [utile]

Le design pattern Builder est utilisé pour construire des objets complexes étape par étape. Il sépare la construction d'un objet de sa représentation, permettant ainsi de créer différentes variations de l'objet en utilisant le même processus de construction. Le Builder pattern comprend un directeur (Director) qui contrôle le processus de construction à l'aide d'un constructeur (Builder) qui crée l'objet final. Cela rend la création d'objets complexes plus flexible, lisible et maintenable.

Diagramme de classe



Example en code PHP

Dans cet exemple, le Builder (**VoitureBuilder**) est utilisé pour créer un objet **Voiture** avec des composants tels que le moteur, la carrosserie et les roues. Il est possible de configurer la **voiture** en ajoutant ces composants via les méthodes du Builder, puis obtenir la description complète de la voiture en utilisant la méthode **getDescription**.

```
<?php
class Voiture
{
```

```php
    private $moteur;
    private $carrosserie;
    private $roues;

    public function setMoteur($moteur)
    {
        $this->moteur = $moteur;
    }

    public function setCarrosserie($carrosserie)
    {
        $this->carrosserie = $carrosserie;
    }

    public function setRoues($roues)
    {
        $this->roues = $roues;
    }

    public function getDescription()
    {
        return "Voiture avec moteur : {$this->moteur}, carrosserie : {$this->carrosserie}, roues : {$this->roues}";
    }
}

class VoitureBuilder
{
    private $voiture;

    public function __construct()
    {
        $this->voiture = new Voiture();
    }

    public function ajouterMoteur($moteur)
    {
        $this->voiture->setMoteur($moteur);
    }

    public function ajouterCarrosserie($carrosserie)
    {
        $this->voiture->setCarrosserie($carrosserie);
    }

    public function ajouterRoues($roues)
    {
        $this->voiture->setRoues($roues);
    }

    public function obtenirVoiture()
    {
        return $this->voiture;
    }
}

// Utilisation du Builder pour construire une voiture
$builder = new VoitureBuilder();
$builder->ajouterMoteur("V8");
$builder->ajouterCarrosserie("SUV");
$builder->ajouterRoues("18 pouces");

$voiture = $builder->obtenirVoiture();

// Affichage de la description de la voiture
echo $voiture->getDescription(); // Affiche "Voiture avec moteur : V8, carrosserie : SUV, roues : 18 pouces"
?>
```

Utilisations

Le design pattern Builder est utilisé dans les cas suivants :

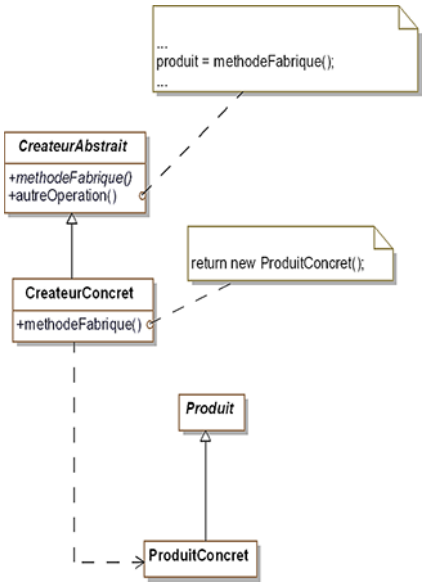
1. Lorsqu'un client souhaite créer des objets complexes sans avoir à se soucier des détails de leur implémentation.
2. Lorsqu'un client a besoin de construire des objets complexes qui peuvent avoir plusieurs représentations ou implémentations différentes.

Factory Method [difficile / utile]

Le design pattern Factory Method est un modèle de conception qui fournit une interface pour créer des objets, mais délègue la responsabilité de l'instanciation aux classes dérivées (sous-classes ou implémentations concrètes). Cela permet de créer des objets de manière polymorphique en laissant chaque sous-classe décider du type d'objet à créer. Le Factory

Method est souvent utilisé pour créer des objets de familles d'objets sans spécifier leur classe exacte lors de la création.

Diagramme de classe



Example en code PHP

Dans cet exemple, le Factory Method (**createDocument**) est utilisé pour créer des documents de différents formats sans connaître les détails de leur création. Cela rend le code plus flexible et évite d'avoir à modifier le code client chaque fois que vous ajoutez un nouveau format de document.

```php
<?php
// Interface pour les documents
interface Document {
    public function genererContenu();
}

// Implémentation de document PDF
class PDFDocument implements Document {
    public function genererContenu() {
        return "Contenu du document PDF";
    }
}

// Implémentation de document HTML
class HTMLDocument implements Document {
    public function genererContenu() {
        return "Contenu du document HTML";
    }
}

// Implémentation de document JSON
class JSONDocument implements Document {
    public function genererContenu() {
        return "Contenu du document JSON";
    }
}

// Factory Method pour créer des documents
class DocumentFactory {
    public function createDocument($format) {
        if ($format === "PDF") {
            return new PDFDocument();
        } elseif ($format === "HTML") {
            return new HTMLDocument();
        } elseif ($format === "JSON") {
            return new JSONDocument();
        } else {
            throw new Exception("Format de document non pris en charge");
        }
    }
}
```

```php
// Utilisation du Factory Method
$factory = new DocumentFactory();

$pdf_document = $factory->createDocument("PDF");
$html_document = $factory->createDocument("HTML");
$json_document = $factory->createDocument("JSON");

echo $pdf_document->genererContenu() . "\n";  // Contenu du document PDF
echo $html_document->genererContenu() . "\n"; // Contenu du document HTML
echo $json_document->genererContenu() . "\n"; // Contenu du document JSON
?>
```

Utilisations

Le design pattern Factory Method sert à résoudre le problème de la création d'objets sans spécifier leur classe exacte lors de la création.

1. Dissociation de la création et de l'utilisation
   Le Factory Method permet de séparer la création d'objets de leur utilisation. Les clients n'ont pas besoin de connaître les détails de la création de l'objet. Ils utilisent simplement le Factory Method pour obtenir l'objet dont ils ont besoin.
2. Polymorphisme
   Le Factory Method permet de créer des objets de sous-classes différentes tout en les traitant de manière polymorphique. Cela signifie qu'il est possible d'avoir différentes implémentations de l'objet (par exemple, différentes races de chiens) et les utiliser de la même manière, car ils partagent une interface commune.
3. Encapsulation des détails de création
   Les détails de création sont encapsulés dans les sous-classes du Factory Method. Cela signifie que vous pouvez modifier la façon dont les objets sont créés sans affecter le code client existant. Par exemple, vous pouvez ajouter de nouvelles sous-classes pour créer de nouveaux types d'objets sans toucher au code existant.
4. Flexibilité et extensibilité
   Le Factory Method rend le code plus flexible et extensible. Si besoin de créer de nouveaux types d'objets à l'avenir, il suffit d'ajouter de nouvelles sous-classes sans modifier le code existant.

Le Factory Method est utile lorsque que l'on souhaite créer des objets de manière flexible et polymorphique tout en cachant les détails de création aux clients du code. Il favorise une conception plus modulaire et maintenable en évitant la dépendance directe des clients aux classes concrètes d'objets.

Les design patterns de structuration

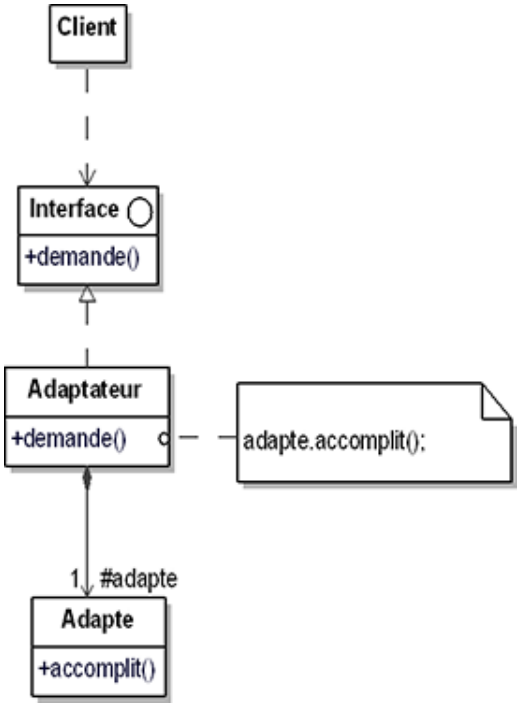
Les **design patterns de structuration**, également appelés design patterns structurels, résolvent généralement des problèmes liés à la manière dont les classes et les objets sont structurés et organisés dans un système logiciel. Ils visent à améliorer la flexibilité, l'efficacité et la facilité de maintenance du code en fournissant des solutions aux problèmes courants de structuration.
Cette catégorie comprend sept modèles : Adapter, Bridge, Composite, Decorator, Facade, Flyweight et Proxy.

Adapter [Facile]

Le design pattern Adapter permet d'adapter une interface d'objet incompatible à une autre interface, permettant ainsi à des objets de collaborer qui ne pourraient pas le faire autrement en

raison de leurs différences d'interfaces. Cela facilite l'interopérabilité entre des classes ou composants ayant des interfaces incompatibles.

Diagramme de classe



Example en code Dart

Dans cet exemple, nous avons deux bibliothèques Bluetooth différentes (**MobileBluetooth** et **DesktopBluetooth**) avec des interfaces spécifiques. En utilisant les adaptateurs (**MobileBluetoothAdapter** et **DesktopBluetoothAdapter**), nous pouvons les adapter à l'interface commune **BluetoothInterface** et les utiliser de la même manière dans notre application, ce qui est simplifié par une interface commune.

```dart
// Interface commune pour Bluetooth
abstract class BluetoothInterface {
  void enable();
  void connect();
}

// Bibliothèque Bluetooth Mobile (interface spécifique)
class MobileBluetooth {
  void start() {
    print("Démarrage de la bibliothèque Bluetooth mobile");
  }

  void pair() {
    print("Appairage avec la bibliothèque Bluetooth mobile");
  }
}

// Bibliothèque Bluetooth Desktop (interface spécifique)
class DesktopBluetooth {
  void powerOn() {
    print("Mise sous tension de la bibliothèque Bluetooth de bureau");
  }

  void establishConnection() {
    print(
        "Établissement de la connexion avec la bibliothèque Bluetooth de bureau");
  }
}

// Adaptateur pour l'ancienne bibliothèque Bluetooth
class MobileBluetoothAdapter implements BluetoothInterface {
  MobileBluetooth _MobileBluetooth;

  MobileBluetoothAdapter(MobileBluetooth MobileBluetooth) {
    _MobileBluetooth = MobileBluetooth;
  }
```

```
  @override
  void enable() {
    _MobileBluetooth.start();
  }

  @override
  void connect() {
    _MobileBluetooth.pair();
  }
}

// Adaptateur pour la nouvelle bibliothèque Bluetooth
class DesktopBluetoothAdapter implements BluetoothInterface {
  DesktopBluetooth _DesktopBluetooth;

  DesktopBluetoothAdapter(DesktopBluetooth DesktopBluetooth) {
    _DesktopBluetooth = DesktopBluetooth;
  }

  @override
  void enable() {
    _DesktopBluetooth.powerOn();
  }

  @override
  void connect() {
    _DesktopBluetooth.establishConnection();
  }
}

// Fonction principale
void main() {
  // Utilisation de la bibliothèque Bluetooth Mobile via l'adaptateur
  final MobileBluetooth = MobileBluetooth();
  final MobileBluetoothAdapter = MobileBluetoothAdapter(MobileBluetooth);

  MobileBluetoothAdapter.enable();
  MobileBluetoothAdapter.connect();

  // Utilisation de la bibliothèque Bluetooth Desktop via l'adaptateur
  final DesktopBluetooth = DesktopBluetooth();
  final DesktopBluetoothAdapter = DesktopBluetoothAdapter(DesktopBluetooth);

  DesktopBluetoothAdapter.enable();
  DesktopBluetoothAdapter.connect();
}
```

Utilisations

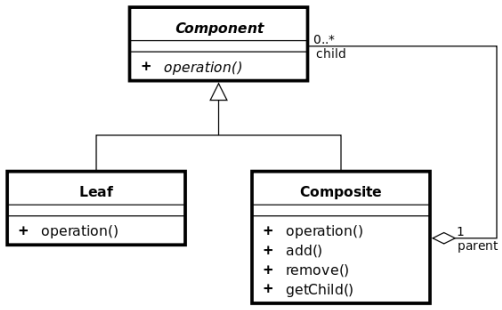
Le design pattern Adapter est appliqué dans les situations suivantes :

1. Quand il est nécessaire d'intégrer un objet dans un système, mais que son interface ne correspond pas à celle requise par ce système.
2. Lorsque l'on souhaite offrir plusieurs interfaces à un objet lors de sa conception, pour le rendre compatible avec différents contextes d'utilisation.

Composite [Utile]

Le design pattern Composite est un modèle de conception qui permet de traiter des objets individuels et des collections d'objets de manière uniforme. Il crée une hiérarchie d'objets, où les objets individuels et les collections d'objets sont traités de la même manière. Cela permet de créer des structures complexes tout en maintenant la simplicité de l'utilisation. Le pattern Composite est souvent utilisé pour représenter des arbres de structures hiérarchiques, où les feuilles et les nœuds internes sont traités de la même manière.

Diagramme de classe



Example en code Java

Les classes **Cercle** et **Carre** qui implémentent l'interface **Forme**, appelé « Composant ». Ensuite, nous avons la classe **GroupeDeFormes**, qui est un « composite » pouvant contenir d'autres formes, y compris d'autres groupes de formes. Lorsque nous appelons la méthode **dessiner()** sur le groupe, il dessine l'ensemble de la hiérarchie de formes, qu'il s'agisse de **feuilles** (cercles et carrés) ou d'autres composites (groupes de formes). Le design pattern Composite permet de traiter uniformément des objets individuels et des collections d'objets dans une structure arborescente.

```java
import java.util.ArrayList;
import java.util.List;

// Composant
interface Forme {
    void dessiner();
}

// Feuille
class Cercle implements Forme {
    @Override
    public void dessiner() {
        System.out.println("Dessiner un cercle");
    }
}

// Feuille
class Carre implements Forme {
    @Override
    public void dessiner() {
        System.out.println("Dessiner un carré");
    }
}

// Composite
class GroupeDeFormes implements Forme {
    private List<Forme> formes = new ArrayList<>();

    public void ajouterForme(Forme forme) {
        formes.add(forme);
    }

    @Override
    public void dessiner() {
        System.out.println("Groupe de formes :");
        for (Forme forme : formes) {
            forme.dessiner();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Cercle cercle = new Cercle();
        Carre carre = new Carre();

        GroupeDeFormes groupe = new GroupeDeFormes();
        groupe.ajouterForme(cercle);
        groupe.ajouterForme(carre);

        groupe.dessiner();

        GroupeDeFormes groupe2 = new GroupeDeFormes();
        groupe2.ajouterForme(groupe);
        groupe2.ajouterForme(new Carre());

        groupe2.dessiner();
    }
```

```
}
```

Utilisations

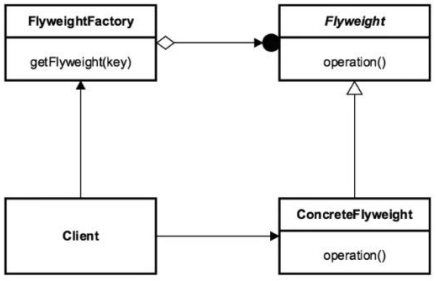
Le design pattern Composite est appliqué dans les situations suivantes :

1. Lorsqu'on veut traiter à la fois des objets individuels et des collections d'objets de manière uniforme au sein d'une structure arborescente.
2. Quand on a besoin de créer des hiérarchies complexes de composants tout en maintenant la simplicité d'utilisation, en permettant aux clients de manipuler des feuilles et des nœuds de la même manière.

Flyweight [Difficile]

Le pattern Flyweight est un modèle de conception structurel pour minimiser la consommation de mémoire en partageant efficacement des objets, notamment en partageant les données intrinsèques entre plusieurs objets similaires. Il vise à réduire la duplication de données et à optimiser les performances en réutilisant des objets, tout en maintenant des objets spécifiques pour les données.

Diagramme de classe



Example en code Java

Le design pattern Flyweight est utilisé pour optimiser le stockage des objets Ville en les partageant. Au lieu de créer une nouvelle instance de **Ville** pour chaque annonce de colocation, la Factory **VilleFactory** gère les objets **Ville** existants et les réutilise si la ville a déjà été créée. Cela permet d'économiser de la mémoire et d'optimiser les performances du site web.

!! Le Hashmap s’assure que la classe est à l’identique

```
import java.util.HashMap;

// Classe pour représenter une ville, appelé Flyweight
class Ville {
    private final String nom;

    public Ville(String nom) {
        this.nom = nom;
    }

    public String getNom() {
        return nom;
    }
}

// Factory pour créer et gérer les objets Ville
class VilleFactory {
    private static final HashMap<String, Ville> villes = new HashMap<>();

    public static Ville getVille(String nom) {
        Ville ville = villes.get(nom);
```

```
        if (ville == null) {
            ville = new Ville(nom);
            villes.put(nom, ville);
        }

        return ville;
    }
}

// Annonce de Colocation
class AnnonceColocation {
    private Ville ville;
    private String description;
    private double loyer;

    public AnnonceColocation(String nomVille, String description, double loyer) {
        this.ville = VilleFactory.getVille(nomVille);
        this.description = description;
        this.loyer = loyer;
    }

    public void afficherAnnonce() {
        System.out.println("Ville : " + ville.getNom());
        System.out.println("Description : " + description);
        System.out.println("Loyer : " + loyer + " CHF par mois");
        System.out.println();
    }
}

public class Main {
    public static void main(String[] args) {
        // Création d'annonces de colocation
        AnnonceColocation annonce1 = new AnnonceColocation("Paris", "Chambre orienté sud", 800);
        AnnonceColocation annonce2 = new AnnonceColocation("Paris", "Appartement magnifique", 1200);
        AnnonceColocation annonce3 = new AnnonceColocation("Grenoble", "Colocation étudiante", 600);

        // Affichage des annonces
        annonce1.afficherAnnonce();
        annonce2.afficherAnnonce();
        annonce3.afficherAnnonce();
    }
}
```

Utilisations

Le design pattern Flyweight est utilisé dans les contextes suivants :

1. Quand il est nécessaire de réduire la consommation de mémoire en partageant efficacement des objets qui ont des données communes, ce qui permet de minimiser la duplication d'informations.
2. Lorsque l'on souhaite offrir plusieurs instances d'objets avec des données intrinsèques partagées, ce qui rend possible leur utilisation dans différents contextes tout en économisant des ressources.

Les design patterns de comportement

Les **design patterns de comportement** fournissent des solutions pour gérer les interactions et répartir les traitements entre les objets.

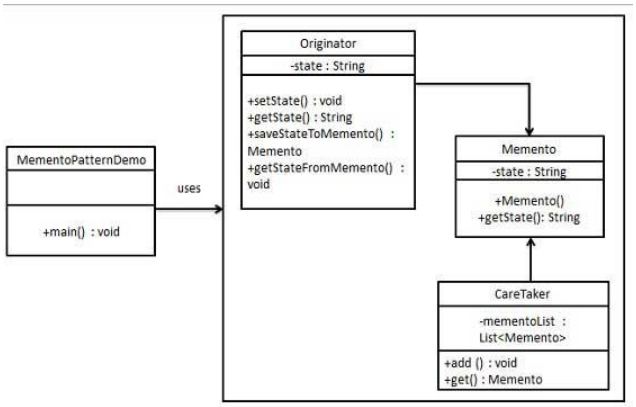
Cette catégorie comprend onze modèles : Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method et Visitor.

Memento [Facile]

Le design pattern Memento est un modèle de conception comportemental qui permet de capturer et de restaurer l'état interne d'un objet sans violer l'encapsulation. Il permet de créer des instantanés de l'état d'un objet à un moment donné, puis de restaurer cet état ultérieurement. Cela est utile pour annuler des actions, gérer l'historique, ou créer des points de sauvegarde dans une application.

Dans le design pattern Memento, on n'enregistre généralement pas directement l'objet en entier, car cela pourrait violer le principe d'encapsulation et poser des problèmes de sécurité. Au lieu de cela, on enregistre uniquement les données internes pertinentes de l'objet, généralement celles qui définissent son état.

Diagramme de classe



Example en code PHP

Cet exemple PHP met en œuvre le design pattern Memento avec trois composants essentiels. Tout d'abord, nous avons **l'Originator**, représenté par la classe **Course**, qui est l'objet principal dont l'état est sauvegardé et restauré. Ensuite, le Memento, incarné par la classe **CourseMemento**, sert à stocker un instantané de l'état de **l'Originator**, dans ce cas, le nom de la course. Enfin, le **CareTaker**, géré par la classe **CourseManager**, joue le rôle de gestionnaire de mementos. Il est chargé de la sauvegarde des mementos et de leur restitution au besoin. Dans l'exemple, le nom de la course peut être modifié dans **l'Originator**, et grâce au Memento, son état est sauvegardé via la méthode "sauvegarder" et restauré via la méthode "restaurer". Cela permet de maintenir un historique des noms de course, offrant une gestion flexible de l'état de l'objet principal.

```
import java.util.HashMap;

// Classe pour représenter une ville, appelé Flyweight
class Ville {
    private final String nom;

    public Ville(String nom) {
        this.nom = nom;
    }

    public String getNom() {
        return nom;
    }
}

// Factory pour créer et gérer les objets Ville
class VilleFactory {
    private static final HashMap<String, Ville> villes = new HashMap<>();

    public static Ville getVille(String nom) {
        Ville ville = villes.get(nom);
```

```
        if (ville == null) {
            ville = new Ville(nom);
            villes.put(nom, ville);
        }

        return ville;
    }
}

// Annonce de Colocation
class AnnonceColocation {
    private Ville ville;
    private String description;
    private double loyer;

    public AnnonceColocation(String nomVille, String description, double loyer) {
        this.ville = VilleFactory.getVille(nomVille);
        this.description = description;
        this.loyer = loyer;
    }

    public void afficherAnnonce() {
        System.out.println("Ville : " + ville.getNom());
        System.out.println("Description : " + description);
        System.out.println("Loyer : " + loyer + " CHF par mois");
        System.out.println();
    }
}

public class Main {
    public static void main(String[] args) {
        // Création d'annonces de colocation
        AnnonceColocation annonce1 = new AnnonceColocation("Paris", "Chambre orienté sud", 800);
        AnnonceColocation annonce2 = new AnnonceColocation("Paris", "Appartement magnifique", 1200);
        AnnonceColocation annonce3 = new AnnonceColocation("Grenoble", "Colocation étudiante", 600);

        // Affichage des annonces
        annonce1.afficherAnnonce();
        annonce2.afficherAnnonce();
        annonce3.afficherAnnonce();
    }
}
```

Utilisations

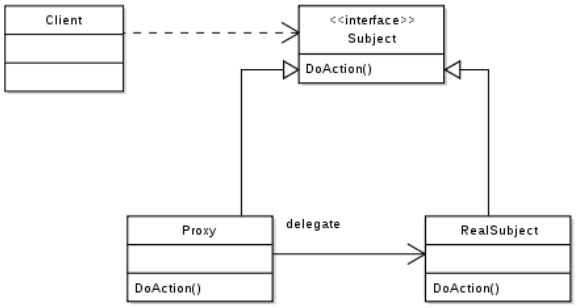
Le design pattern Memento est employé dans les situations suivantes :

1. Quand il est impératif de capturer et de restaurer l'état d'un objet pour économiser de la mémoire en partageant efficacement des données communes, réduisant ainsi la duplication d'informations.
2. Lorsque l'objectif est de fournir la possibilité d'enregistrer et de revenir à plusieurs états antérieurs d'un objet, ce qui permet d'explorer différentes versions tout en préservant les ressources.

Proxy [Utile]

Le design pattern Proxy est un modèle de conception structurel qui permet de contrôler l'accès à un objet en agissant comme une interface intermédiaire. Il est utilisé pour ajouter des fonctionnalités telles que la gestion de la sécurité, le « chargement paresseux – lazy loading», la mise en cache, ou la gestion des appels distants tout en maintenant la même interface que l'objet réel.

Diagramme de classe



Example en code Java

Dans cet exemple, le Proxy **AnnonceProxy** contrôle l'accès aux détails de l'annonce. Les détails réels ne sont chargés qu'au moment de l'accès, illustrant le concept de chargement paresseux avec un proxy.

```
import java.util.ArrayList;
import java.util.List;

// Interface pour les annonces
interface Annonce {
    String getTitle();
    String getDescription();
}

// Classe réelle qui représente une annonce
class AnnonceReelle implements Annonce {
    private String title;
    private String description;

    public AnnonceReelle(String title, String description) {
        this.title = title;
        this.description = description;
    }

    public String getTitle() {
        return title;
    }

    public String getDescription() {
        return description;
    }
}

// Classe proxy pour les annonces
class AnnonceProxy implements Annonce {
    private Annonce annonceReelle = null;
    private String title;
    private String description;

    public AnnonceProxy(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }

    public String getDescription() {
        if (annonceReelle == null) {
            annonceReelle = new AnnonceReelle(title, description);
        }
        return annonceReelle.getDescription();
    }

    public void setDescription(String description) {
        this.description = description;
    }
}

public class Main {
    public static void main(String[] args) {
        List<Annonce> annonces = new ArrayList<>();
        AnnonceProxy annonce1 = new AnnonceProxy("Annonce 1");
        annonces.add(annonce1);

        // Les détails ne sont chargés qu'au moment de l'accès
        annonce1.setDescription("Description de l'annonce 1");

        System.out.println("Liste des Annonces de Colocation :");
        for (Annonce annonce : annonces) {
```

```
        System.out.println("Titre : " + annonce.getTitle());
        System.out.println("Description : " + annonce.getDescription()); //ICI
        System.out.println();
    }
  }
}
```

Utilisations

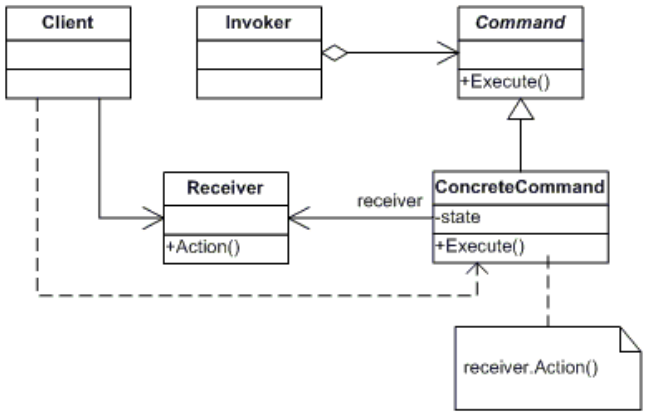
Le design pattern Proxy est utilisé dans les contextes suivants :

1. Lorsqu'il est nécessaire de contrôler l'accès à un objet en agissant comme une interface intermédiaire, permettant d'ajouter des fonctionnalités telles que la gestion de la sécurité, le chargement paresseux, la mise en cache, ou la gestion des appels distants tout en maintenant la même interface que l'objet réel.
2. Quand on souhaite retarder ou optimiser le chargement de ressources coûteuses (par exemple, images, vidéos) jusqu'à ce qu'elles soient réellement nécessaires, améliorant ainsi les performances de l'application en évitant le chargement inutile de ressources.

Command [Difficile]

Le design pattern Command est un modèle de conception comportemental qui encapsule une requête en tant qu'objet, permettant de paramétrer, exécuter, annuler et mettre en file d'attente des demandes. Il sépare l'émetteur d'une demande (le client) de l'objet qui effectue l'action (le récepteur), offrant ainsi plus de flexibilité, de déclenchement d'actions et de gestion d'opérations réversibles.

Diagramme de classe



Example en code PHP

Ce code utilise le design pattern Command pour contrôler une lampe à l'aide d'une télécommande. Une interface **Action** est créée pour représenter une action, et des classes **ActionAllumer** et **ActionEteindre** implémentent cette interface pour les actions d'allumage et d'extinction de la lampe. La classe **Lampe** est le récepteur, tandis que la classe **Telecommande** est l'invocateur qui permet de définir et d'exécuter les actions. L'exemple montre comment ce pattern permet de déconnecter l'émetteur de l'action du récepteur en encapsulant les actions dans des objets, offrant ainsi une manière flexible de gérer les actions.

```php
<?php
// Interface pour l' action
interface Action {
    public function executer();
}

// Récepteur - La lampe
class Lampe {
    public function allumer() {
        echo "La lampe est allumée.";
    }

    public function eteindre() {
        echo "La lampe est éteinte.";
    }
}

// Action pour allumer la lampe
class ActionAllumer implements Action {
    private $lampe;

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

    public function executer() {
        $this->lampe->allumer();
    }
}

// Action pour éteindre la lampe
class ActionEteindre implements Action {
    private $lampe;

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

    public function executer() {
        $this->lampe->eteindre();
    }
}

// Invocateur - La télécommande
class Telecommande {
    private $Action;

    public function setAction(Action $Action) {
        $this->Action = $Action;
    }

    public function boutonPresse() {
        $this->Action->executer();
    }
}

// Utilisation
$lampe = new Lampe();
$ActionAllumer = new ActionAllumer($lampe);
$ActionEteindre = new ActionEteindre($lampe);

$telecomande = new Telecommande();
$telecomande->setAction($ActionAllumer);
$telecomande->boutonPresse();

$telecomande->setAction($ActionEteindre);
$telecomande->boutonPresse();
```

Utilisations

Le design pattern Command est utilisé dans les contextes suivants :

1. Lorsqu'il est essentiel de séparer l'émetteur d'une commande (le client) du récepteur de l'action (l'objet qui effectue l'action), ce qui permet une gestion plus flexible des opérations tout en évitant la duplication d'informations et en économisant des ressources.
2. Quand on souhaite encapsuler des actions dans des objets, les paramétrer, les mettre en file d'attente, les annuler et les réexécuter, offrant ainsi une gestion réversible des opérations et la possibilité d'explorer différentes versions d'un système.

Conclusion générale

En conclusion, les design patterns représentent une ressource essentielle dans le domaine du développement logiciel. Ils sont le fruit de nombreuses années d'expérience et de réflexion collective de la part de développeurs. Ces modèles offrent des solutions éprouvées pour des problèmes récurrents, favorisent la réutilisation du code, améliorent la maintenabilité, et aident à concevoir des systèmes robustes et évolutifs.

L'utilisation judicieuse des design patterns permet aux développeurs de créer des logiciels plus efficaces, plus lisibles et plus faciles à entretenir. Ils offrent un langage commun pour communiquer des idées de conception, ce qui simplifie la collaboration au sein des équipes de développement.

Toutefois, il est essentiel de se rappeler que les design patterns ne sont pas une solution miracle. Ils ne doivent pas être appliqués de manière aveugle, mais plutôt adaptés à chaque contexte spécifique. La compréhension des principes sous-jacents qui motivent chaque pattern est essentielle pour choisir le design pattern approprié et l'implémenter de manière adéquate.

Conclusion personnelle

Je me suis trouvé face à un livre qui constitue une ressource précieuse pour les concepteurs et développeurs orientés objet. Il offre une présentation claire et pratique des 23 modèles de conception fondamentaux, en les illustrant avec des exemples concrets en UML2 et en PHP 8.

Pour bien l'appréhender, il faut avoir des connaissances en UML et en langage de programmation POO. Il va droit au but, en abordant immédiatement la problématique et en présentant la solution sans détours ni discours superflu.
Les cas pratiques sont présentés de manière méthodique, même si je n'aurai pas l'occasion de les rencontrer tous au cours de ma pratique.