Référentiel de compétences

Semestre 1

Module

Semestre 2

Artéfact

Session de formation

Business plan

Lecture individuelle

Systèmes d'information & management des organisations

Docker

Métodologie en cascade

La veille informatique

Les design patterns

> *Le but de ce document est de renforcer les connaissances en programmation orientée objet. Découvrir les design patterns et les mettre en pratique.*

Design pattern veut dire : patrons de conception ou schéma de conception. Ce sont des solutions à des problèmes récurrents en programmation.

C'est une solution réutilisable à un problème courant rencontré lors de la conception de logiciels. Il s'agit d'une approche éprouvée et documentée pour résoudre des problèmes de conception spécifiques dans le développement logiciel.

Les design patterns fournissent des modèles de conception génériques qui peuvent être adaptés à différentes situations. Ils permettent de capturer l'expertise et l'expérience accumulées dans le domaine du développement logiciel et de les appliquer de manière cohérente pour résoudre des problèmes similaires.

Dans quel cas utiliser un design pattern / schéma de conception ?

Une application évolue au fil du temps. Lorsqu'elle est développée, de nouvelles fonctionnalités sont ajoutées et cela peut rendre le code source plus complexe. Elle devient dificile à maintenir et à contrôler.

On va devoir refactoriser le code !

> *La factorisation, c'est le principe de modifier le code source de manière à ce que le même code soit utilisé à plusieurs endroits. Cela permet de réduire la taille du code source et de le rendre plus facile à maintenir. Et tout cela sans ajouter de nouvelles fonctionalités.*

Comment faire proprement du refactoring ? En utilisant des design patterns !

Donc les design patterns sont des solutions type à des problèmes spécifiques.

Il existe trois grands types de design patterns :

- Les design patterns de création sont utilisés pour instancier des objets ou des groupes d'objets liés.
- Les design patterns de structure sont utilisés pour former de grandes structures de classe à partir de classes individuelles.
- Les design patterns de comportement sont utilisés pour gérer la communication entre les objets.

Ne pas confondre

- Un design pattern est relatif à la création, la manipulation et/ou la communication entre un ou plusieurs objets. Dans ce cas, vous travaillez sur un sujet assez précis : la communication avec une

API, par exemple.

- Un design d’architecture est un ensemble de règles et de conventions pour le design de l’application elle-même, qu’elle soit front-end ou back-end… Par exemple, quelle partie du code s’occupe de gérer les données, quelle partie s’occupe de les afficher, etc.
- Un framework est une solution clé en main, comme une boîte à outil pour la création d'un service spécifique (WebApp, API, etc)

Creation Design Patterns

Constructor Pattern

> *Il s'agit du pattern que vous devriez normalement connaître !*

Le but du Constructor Pattern est de :

- **Formater** les données
- **Créer** des objets

```
public class Person {
    private String firstName;
    private String lastName;
    private int age;
    private String address;

    public Person(String firstName, String lastName, int age, String address) {
```

```
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.address = address;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public int getAge() {
        return age;
    }

    public String getAddress() {
        return address;
    }
}

Person person = new Person("John", "Doe", 42, "1 rue de la Paix");
```

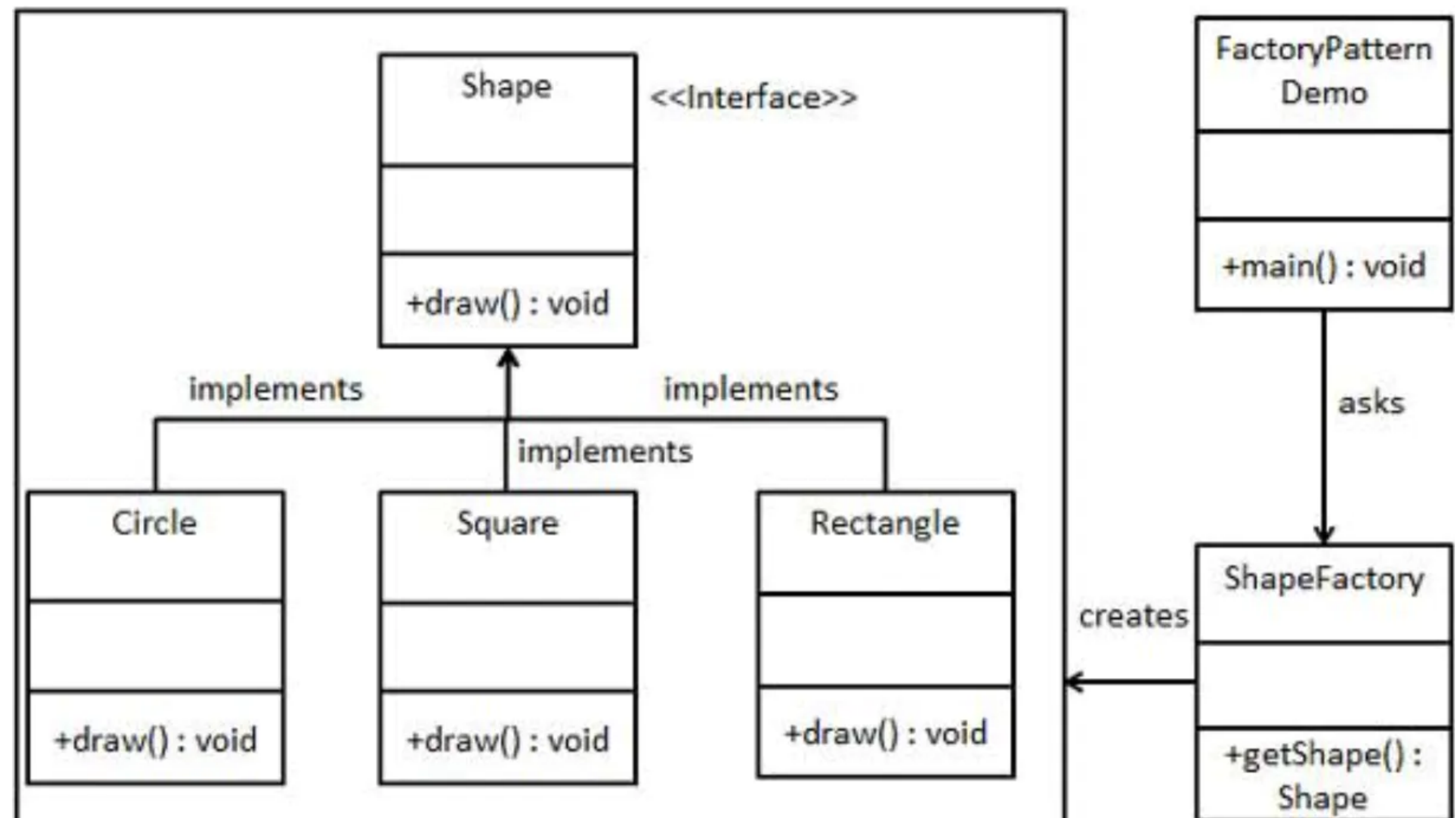
Factory Pattern

Le Factory Pattern est utilisé pour déléguer la création d'objets. Au lieu de créer vos objets "en direct", vous passez par un autre objet (la Factory) qui va se charger de créer le bon objet.

Quand utiliser ce pattern ?

- Lorsque la création d'un objet implique une logique complexe ou conditionnelle.
- Lorsque vous voulez centraliser la création d'objets dans une seule classe ou méthode.
- Lorsque vous souhaitez fournir une interface commune pour la création d'objets.

Exemple lorsque l'on ajoute à une application une dexuième source de données externe et que le format change.




```
// Classe parente
abstract class Animal {
    public abstract void makeSound();
```

```
}

// Sous-classes d'Animal
class Dog extends Animal {
    public void makeSound() {
        System.out.println("Le chien aboie !");
    }
}

class Cat extends Animal {
    public void makeSound() {
        System.out.println("Le chat miaule !");
    }
}

// Factory pour la création d'animaux
class AnimalFactory {
    public static Animal createAnimal(String animalType) {
        if (animalType.equalsIgnoreCase("dog")) {
            return new Dog();
        } else if (animalType.equalsIgnoreCase("cat")) {
            return new Cat();
        } else {
            throw new IllegalArgumentException("Type d'animal non pris en charge : " + animalType);
        }
    }
}

// Utilisation du Factory Pattern
public class Main {
    public static void main(String[] args) {
        Animal dog = AnimalFactory.createAnimal("dog");
        dog.makeSound(); // Affiche "Le chien aboie !"

        Animal cat = AnimalFactory.createAnimal("cat");
```

```
        cat.makeSound(); // Affiche "Le chat miaule !"
    }
}
```

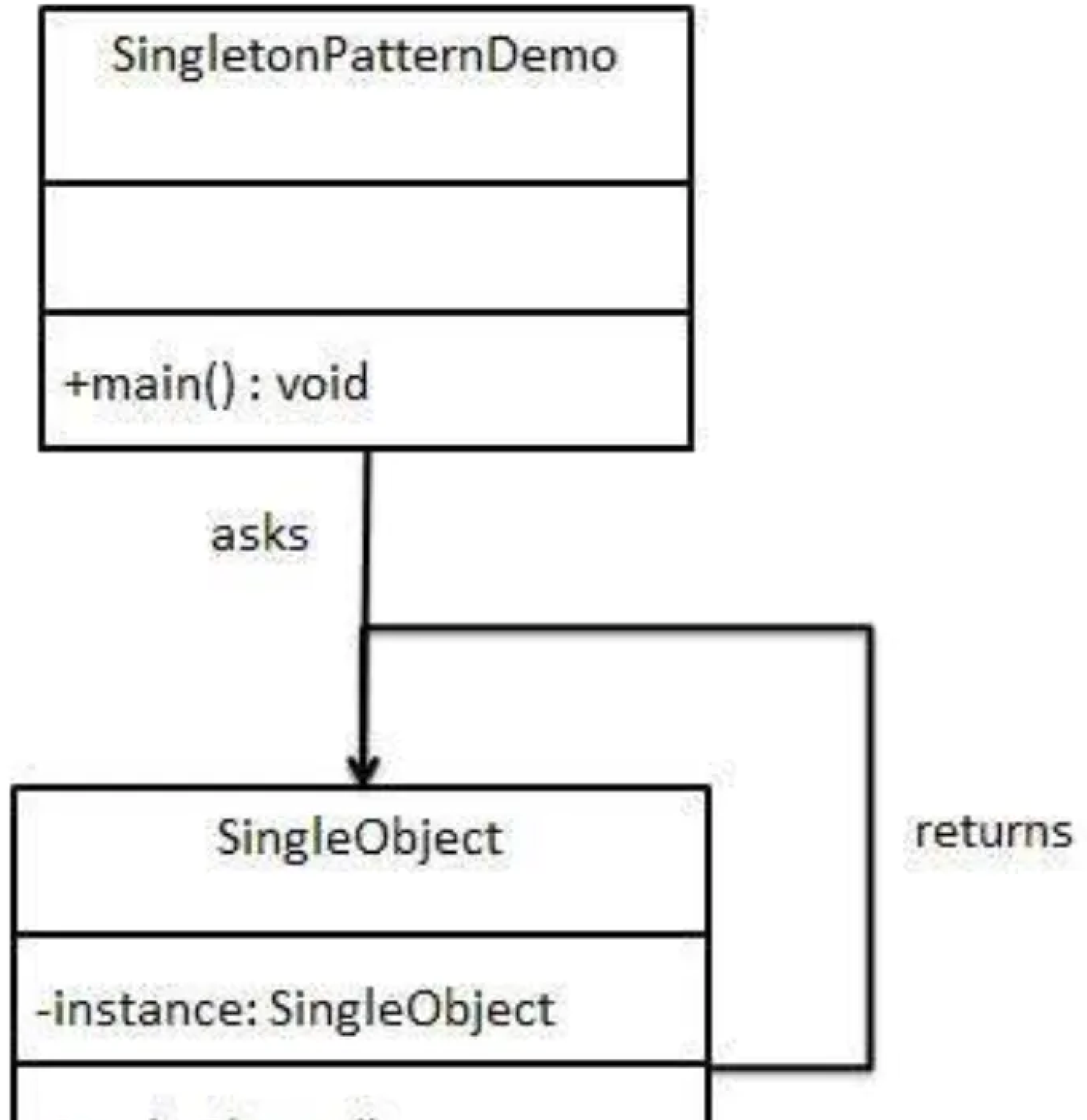
Singleton Pattern

Le Singleton Pattern est utilisé pour s'assurer qu'une classe n'a qu'une seule instance et pour fournir un point d'accès global à cette instance. Il permet une écnomie des ressources !

Quand utiliser ce pattern ?

- Gestion des ressources partagées : Si vous avez une ressource unique, telle qu'une base de données ou une file d' attente, le Singleton peut être utilisé pour garantir qu'une seule instance de cette ressource est créée et partagée entre différentes parties du programme.
- Configuration globale : Si vous avez des paramètres de configuration qui doivent être accessibles à partir de différentes parties de votre application, le Singleton peut être utilisé pour stocker ces informations et permettre un accès centralisé.
- Contrôle de l'accès à une ressource partagée : Si vous avez une ressource critique qui ne peut être utilisée que par une seule entité à la fois, le Singleton peut être utilisé pour contrôler l'accès à cette ressource et éviter les conflits.

Un cas d'utilisation concret, s'utilise plûtot en Back-end, pour la connexion à une base de données par exemple.


SingletonPatternDemo
+main() : void
asks
returns
SingleObject
-instance: SingleObject

-SingleObject ()
+getInstance():SingleObject
+showMessage():void

```java
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}

public class Main {
    public static void main(String[] args) {
        Singleton singleton = Singleton.getInstance();
        singleton.showMessage();
    }
}
```

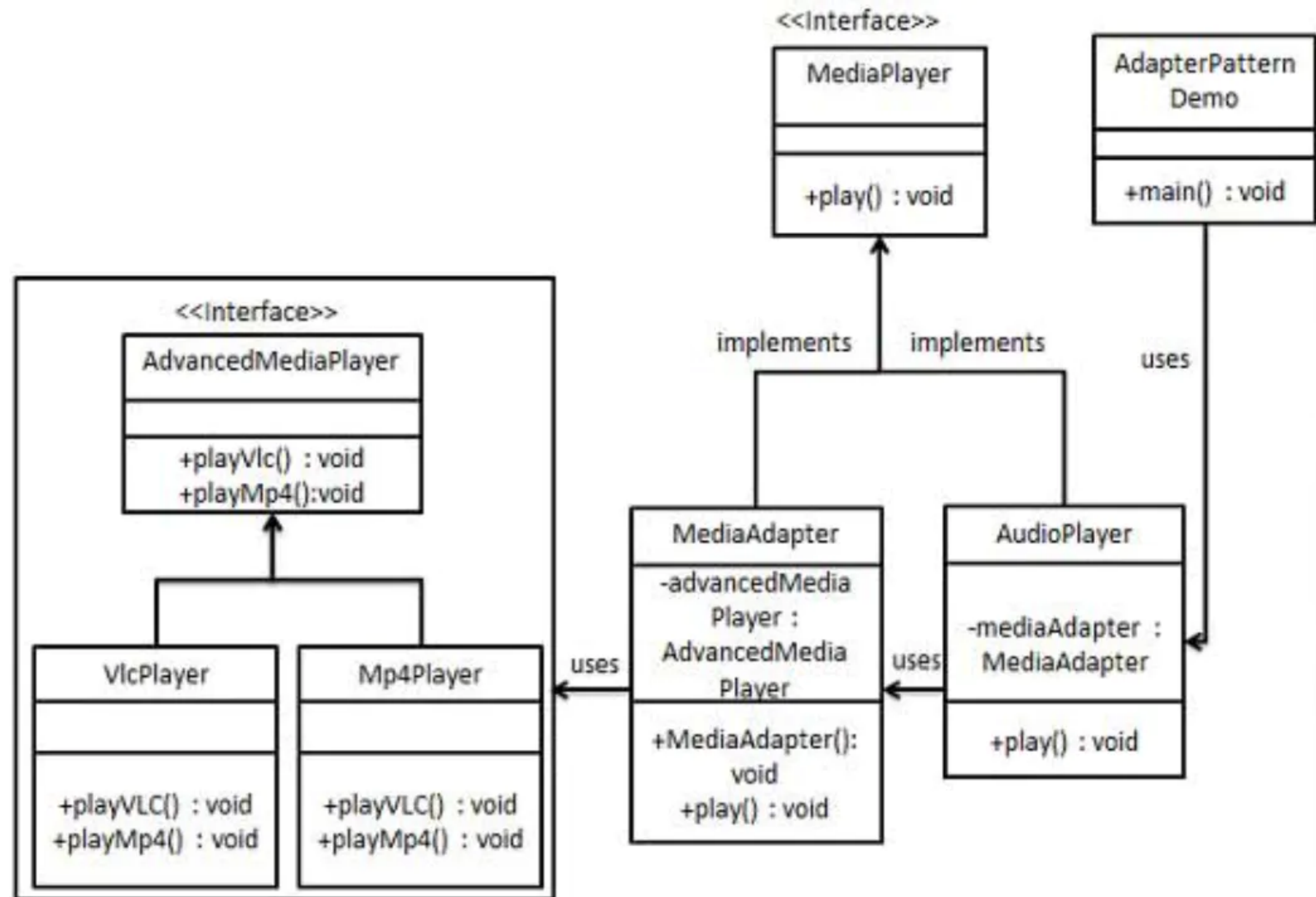
Structural Design Patterns

Adapter Pattern

Le Adapter Pattern est utilisé pour convertir l'interface d'une classe en une autre interface que le client attend. Il permet à des classes qui n'auraient pas pu travailler ensemble de le faire.

Quand l'utiliser ?

- Intégration de nouveaux composants : Lorsque vous souhaitez utiliser un composant existant dans votre système qui a une interface incompatible avec le reste du code. L'adapter permet de créer une interface commune pour le nouveau composant, facilitant ainsi son utilisation.
- Réutilisation de code existant : Lorsque vous avez du code existant qui offre une fonctionnalité précieuse, mais dont l'interface ne correspond pas exactement à ce dont vous avez besoin. L'adapter peut envelopper ce code existant et le rendre utilisable avec l'interface requise.
- Migration de code : Lorsque vous effectuez une mise à jour ou une migration dans votre système, l'adapter peut être utilisé pour maintenir la compatibilité avec le code existant tout en introduisant de nouvelles interfaces.



Voici la situation initale :

```
// Interface cible
interface Cible {
    void requete();
}

// Classe existante sans l'adaptateur
```

```java
class ClasseExistante {
    void operationExistante() {
        System.out.println("Opération existante appelée.");
    }
}

// Utilisation de la classe existante sans l'adaptateur
public class MainWithoutAdapter {
    public static void main(String[] args) {
        ClasseExistante classeExistante = new ClasseExistante();
        classeExistante.operationExistante();
    }
}
```

Maintenant l'api a changé et on doit utiliser un adaptateur parce qu'on ne peut pas changer la classe existante (ClasseExistante), parce qu'on a pas le contrôle dessus.

```java
// Interface cible
interface Cible {
    void requete();
}

// Classe existante (l'adapté)
class ClasseExistante {
    void operationExistante() {
        System.out.println("Opération existante appelée.");
    }
}

// Adaptateur
class Adaptateur implements Cible {
    private ClasseExistante classeExistante;
```

```java
    Adaptateur(ClasseExistante classeExistante) {
        this.classeExistante = classeExistante;
    }

    @Override
    public void requete() {
        classeExistante.operationExistante();
    }
}

// Utilisation de l'adaptateur
public class MainWithAdapter {
    public static void main(String[] args) {
        ClasseExistante classeExistante = new ClasseExistante();
        Cible adaptateur = new Adaptateur(classeExistante);
        adaptateur.requete();
    }
}
```

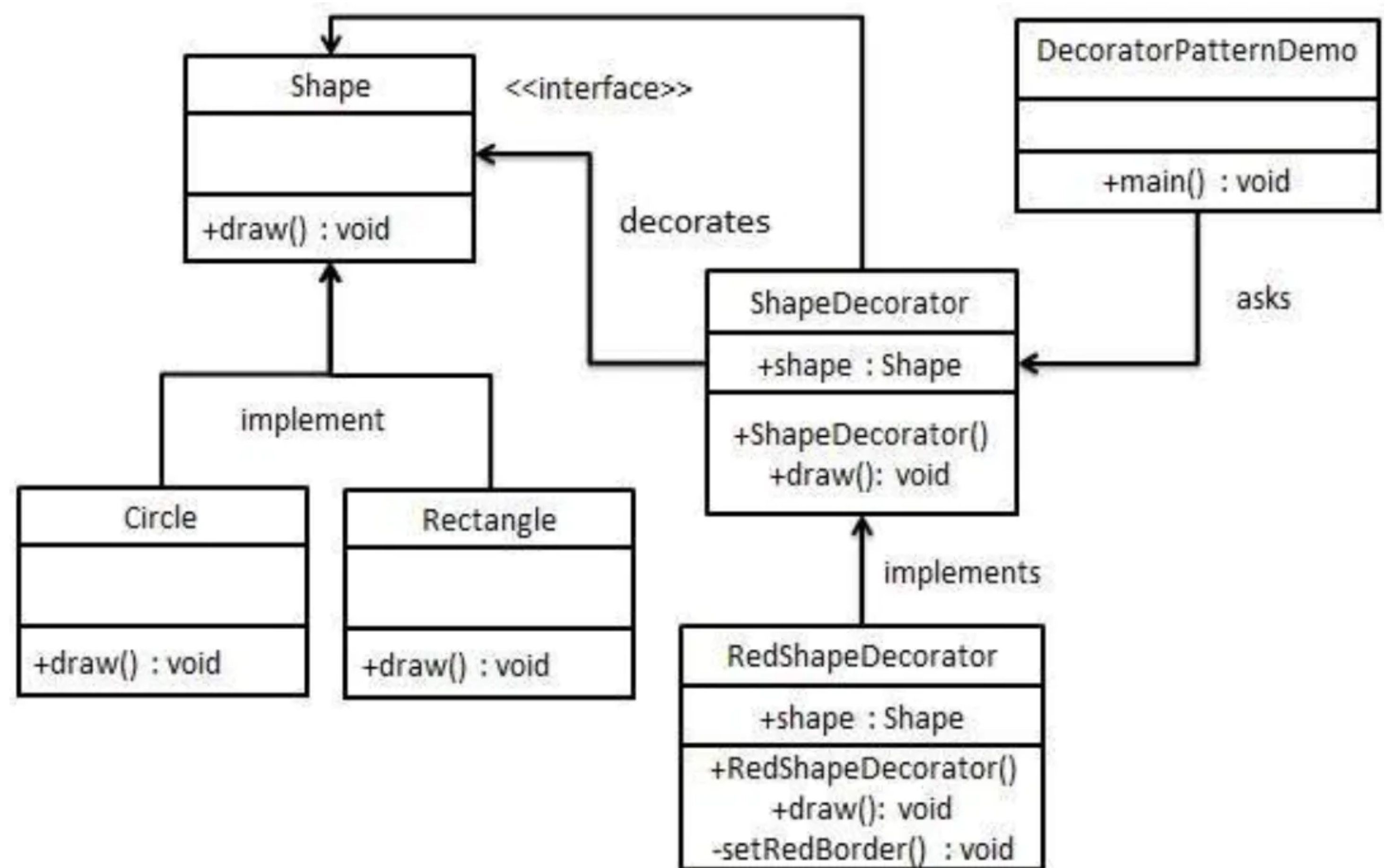
Decorator Pattern

Le Decorator Pattern est utilisé pour ajouter des fonctionnalités à une classe sans modifier le code existant de la classe. Il permet d'ajouter des fonctionnalités à une méthode sans modifier le code de la méthode et la classe.

Quand l'utiliser ?

- Ajouter des fonctionnalités supplémentaires à un objet sans modifier son code source existant.
- Étendre les capacités d'un objet de manière flexible et modulaire.

- Combiner plusieurs fonctionnalités ou comportements de manière dynamique.
- Permettre la composition d'objets avec des comportements différents à l'exécution.
- Éviter une classe avec une hiérarchie de sous-classes trop complexe et rigide.



Supposons que nous ayons une interface Composant représentant un élément de base avec une méthode operation().

```
public interface Composant {
    void operation();
}
```

Ensuite, nous avons une implémentation concrète de cette interface appelée ComposantConcret, qui représente l'objet de base que nous souhaitons décorer.

```
public class ComposantConcret implements Composant {
    @Override
    public void operation() {
        System.out.println("Opération de l'objet de base.");
    }
}
```

Maintenant, nous allons créer un décorateur abstrait appelé Decorateur qui implémente également l'interface Composant et a une référence à un objet de type Composant. Cela nous permettra d'ajouter des fonctionnalités supplémentaires.

```
public abstract class Decorateur implements Composant {
protected Composant composant;

    public Decorateur(Composant composant) {
        this.composant = composant;
    }

    @Override
    public void operation() {
        composant.operation();
    }
```

```
}
```

Ensuite, nous pouvons créer des classes concrètes de décorateur qui étendent la classe Decorateur et ajoutent des fonctionnalités supplémentaires.

```java
public class DecorateurConcret extends Decorateur   {
    public DecorateurConcret(Composant composant) {
        super(composant);
    }

    @Override
    public void operation() {
        super.operation();
        ajouterFonctionnaliteSupplementaire();
    }

    private void ajouterFonctionnaliteSupplementaire() {
        System.out.println("Fonctionnalité supplémentaire ajoutée par le décorateur.");
    }
}
```

Enfin, nous pouvons utiliser ces classes pour décorer l'objet de base avec des fonctionnalités supplémentaires.

```java
public class ExempleUtilisation {
    public static void main(String[] args) {
        Composant composant = new ComposantConcret();
        composant.operation(); // Sortie : "Opération de l'objet de base."

        Composant composantDecore = new DecorateurConcret(new ComposantConcret());
```

```
        composantDecore.operation();
        // Sortie :
        // "Opération de l'objet de base."
        // "Fonctionnalité supplémentaire ajoutée par le décorateur."
    }
}
```

Dans cet exemple, nous avons créé un objet ComposantConcret de base, puis nous l'avons enveloppé avec un DecorateurConcret pour ajouter une fonctionnalité supplémentaire. Lorsque nous appelons la méthode operation() sur l'objet décoré, il exécute d'abord l'opération de l'objet de base, puis ajoute la fonctionnalité supplémentaire spécifique au décorateur.

Cela illustre comment le pattern Decorator permet d'ajouter des fonctionnalités dynamiquement en enveloppant des objets avec des décorateurs.

Proxy pattern

Le pattern Proxy est utilisé pour fournir un substitut ou un espace réservé pour un autre objet afin de contrôler l'accès à celui-ci. Ce qu'on appel aussi un cache.

Quand l'utiliser ?

- Contrôle d'accès : Le Proxy permet de restreindre l'accès à un objet, en vérifiant les autorisations ou en effectuant des validations supplémentaires avant de permettre l'appel à ses méthodes.
- Mise en cache : Le Proxy peut mettre en cache les résultats d'appels coûteux à un objet et les retourner directement lors d'appels ultérieurs, sans invoquer à nouveau l'objet réel.

- Chargement paresseux (Lazy loading) : Le Proxy permet de différer le chargement d'un objet lourd jusqu'à ce qu'il soit réellement nécessaire. Ainsi, il permet d'améliorer les performances en évitant un chargement prématuré.
- Logging et journalisation : Le Proxy peut être utilisé pour enregistrer les appels effectués sur un objet, ce qui permet de suivre l'historique des appels ou de générer des journaux.
- Communication distante : Le Proxy peut agir comme une interface distante pour un objet situé sur une machine distante, en masquant les détails de la communication réseau et en fournissant une interface locale.

```
// Interface du sujet
interface Sujet {
    void effectuerAction();
}

// Sujet concret
class SujetConcret implements Sujet {
    public void effectuerAction() {
        System.out.println("Action effectuée.");
    }
}

// Proxy
class Proxy implements Sujet {
    private SujetConcret sujet;

    public void effectuerAction() {
        if (sujet == null) {
            sujet = new SujetConcret();
        }
```

```
        // Avant d'appeler la méthode de l'objet réel, effectuer des actions supplémentaires si néce
        System.out.println("Avant l'action...");

        // Appeler la méthode de l'objet réel
        sujet.effectuerAction();

        // Après l'appel à la méthode de l'objet réel, effectuer des actions supplémentaires si néce
        System.out.println("Après l'action...");
    }
}

// Utilisation
public class Main {
    public static void main(String[] args) {
        Sujet sujet = new Proxy();
        sujet.effectuerAction();
    }
}
```

Behavior Design Patterns

Observer pattern

Le pattern Observer est utilisé lorsque nous voulons être informés des changements d'état d'un objet. Il définit une relation d'un-à-plusieurs entre les objets, de sorte que lorsqu'un objet change d'état, tous ses observateurs sont notifiés et mis à jour automatiquement.

Quand l'utiliser ?

- Notifications d'événements : Lorsque vous avez besoin de notifier plusieurs objets lorsque qu'un événement se produit, vous pouvez utiliser le pattern Observer. Par exemple, dans une application de messagerie, vous pouvez avoir plusieurs observateurs (utilisateurs) qui doivent être notifiés lorsqu'un nouveau message est reçu.
- Mises à jour de l'interface utilisateur : Lorsque vous souhaitez mettre à jour dynamiquement l'interface utilisateur en fonction des changements d'état d'un objet, vous pouvez utiliser le pattern Observer. Par exemple, dans une application météo, vous pouvez avoir un observateur qui met à jour l'affichage des températures chaque fois que les données météorologiques changent.
- Synchronisation de données : Lorsque vous avez des objets qui doivent être maintenus synchronisés avec un objet source, vous pouvez utiliser le pattern Observer. Par exemple, dans une application de traitement de texte collaboratif, vous pouvez avoir plusieurs observateurs qui doivent être informés lorsque le document principal est modifié.

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

interface StockObserver {
    void onStockChange(String stockName, double price);
}

class StockMarket {
    private List<StockObserver> observers = new ArrayList<>();
    private String stockName;
    private double price;

    public void addObserver(StockObserver observer) {
        observers.add(observer);
    }
```

```
    public void removeObserver(StockObserver observer) {
        observers.remove(observer);
    }

    public void setStockPrice(String stockName, double price) {
        this.stockName = stockName;
        this.price = price;
        notifyObservers();
    }

    private void notifyObservers() {
        for (StockObserver observer : observers) {
            observer.onStockChange(stockName, price);
        }
    }
}

class StockDisplay implements StockObserver {
    private String name;

    public StockDisplay(String name) {
        this.name = name;
    }

    @Override
    public void onStockChange(String stockName, double price) {
        System.out.println(name + " : Le prix de l'action " + stockName + " est maintenant de " + pr
    }
}

public class Main {
    public static void main(String[] args) {
        StockMarket stockMarket = new StockMarket();
```

```
        StockDisplay display1 = new StockDisplay("Affichage 1");
        StockDisplay display2 = new StockDisplay("Affichage 2");

        stockMarket.addObserver(display1);
        stockMarket.addObserver(display2);

        stockMarket.setStockPrice("GOOG", 1450.75);

        stockMarket.removeObserver(display2);

        stockMarket.setStockPrice("AAPL", 250.60);
    }
}
```

State pattern

Le pattern State est utilisé lorsque le comportement d'un objet change en fonction de son état. Il permet de déléguer le comportement à un objet d'état qui représente l'état actuel de l'objet.

Quand l'utiliser ?

Lorsque vous avez un objet qui peut se trouver dans différents états et que son comportement doit varier en fonction de ces états. Par exemple, un processus de commande en ligne qui peut être dans les états "nouvelle commande", "en cours de traitement" ou "expédiée", avec des actions différentes disponibles à chaque étape.

Lorsque vous avez un code avec plusieurs blocs if/else ou switch/case qui vérifient l'état d'un objet et déclenchent différentes actions en fonction de cet état. Le pattern d'état peut vous aider à éviter cette

duplication de code en encapsulant le comportement spécifique à chaque état dans des classes séparées.

Lorsque vous souhaitez faciliter l'ajout de nouveaux états et transitions dans un objet sans avoir à modifier son code existant. Le pattern d'état permet de rendre l'ajout de nouveaux états plus modulaire et facilite l'extension de l'objet sans le rendre fragile.

```
// Interface représentant un état de la machine à café
interface EtatMachineCafe {
    void insererPiece();
    void annulerCommande();
    void boutonCafe();
    void boutonEauChaude();
}

// Implémentation d'un état : Machine prête à recevoir de l'argent
class EtatAttentePiece implements EtatMachineCafe {
    public void insererPiece() {
        System.out.println("Pièce insérée. Veuillez sélectionner votre boisson.");
    }

    public void annulerCommande() {
        System.out.println("Aucune commande à annuler.");
    }

    public void boutonCafe() {
        System.out.println("Veuillez insérer une pièce d'abord.");
    }

    public void boutonEauChaude() {
        System.out.println("Veuillez insérer une pièce d'abord.");
    }
```

```
}

// Implémentation d'un état : Machine en train de préparer du café
class EtatPreparationCafe implements EtatMachineCafe {
    public void insererPiece() {
        System.out.println("Impossible d'insérer une pièce pendant la préparation du café.");
    }

    public void annulerCommande() {
        System.out.println("Annulation de la préparation du café.");
    }

    public void boutonCafe() {
        System.out.println("Le café est en cours de préparation. Veuillez patienter.");
    }

    public void boutonEauChaude() {
        System.out.println("Le café est en cours de préparation. Veuillez patienter.");
    }
}

// Classe représentant la machine à café
class MachineCafe {
    private EtatMachineCafe etat;

    public MachineCafe() {
        // L'état initial est "Attente de pièce"
        etat = new EtatAttentePiece();
    }

    public void setEtat(EtatMachineCafe etat) {
        this.etat = etat;
    }

    // Méthodes de la machine à café qui délèguent les actions à l'état courant
```

```
    public void insererPiece() {
        etat.insererPiece();
    }

    public void annulerCommande() {
        etat.annulerCommande();
    }

    public void boutonCafe() {
        etat.boutonCafe();
    }

    public void boutonEauChaude() {
        etat.boutonEauChaude();
    }
}

// Exemple d'utilisation
public class Main {
    public static void main(String[] args) {
        MachineCafe machineCafe = new MachineCafe();

        machineCafe.insererPiece(); // Sortie : "Pièce insérée. Veuillez sélectionner votre boisson
        machineCafe.boutonCafe();   // Sortie : "Veuillez insérer une pièce d'abord."

        machineCafe.boutonEauChaude();  // Sortie : "Veuillez insérer une pièce d'abord."
        machineCafe.annulerCommande();  // Sortie : "Aucune commande à annuler."

        machineCafe.insererPiece(); // Sortie : "Pièce insérée. Veuillez sélectionner votre boisson
        machineCafe.boutonCafe();   // Sortie : "Le café est en cours de préparation. Veuillez pati

        machineCafe.setEtat(new EtatPreparationCafe());
        machineCafe.insererPiece(); // Sortie : "Impossible d'insérer une pièce pendant la préparat
        machineCafe.annulerCommande();  // Sortie : "Annulation de la préparation du café."
    }
```

```
}
```

Template pattern

Le pattern Template (ou modèle) est utilisé lorsque vous avez une structure d'algorithme commune, mais avec certaines étapes spécifiques qui peuvent varier d'une implémentation à l'autre. Ce pattern est utile lorsque vous voulez éviter la duplication de code tout en permettant aux sous-classes de redéfinir certaines parties spécifiques de l'algorithme.

Quand l'utiliser ?

- Définir le squelette d'un algorithme ou d'un processus, en laissant les détails spécifiques aux sous-classes.
- Réduire la duplication de code en regroupant les parties communes dans une classe de base abstraite.
- Fournir une flexibilité pour les sous-classes afin de personnaliser certaines étapes de l'algorithme sans changer sa structure globale.

```
abstract class Beverage {
    public final void prepare() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
```

```
    private void boilWater() {
        System.out.println("Boiling water");
    }

    abstract void brew();

    private void pourInCup() {
        System.out.println("Pouring into cup");
    }

    abstract void addCondiments();
}

class Coffee extends Beverage {
    void brew() {
        System.out.println("Brewing coffee");
    }

    void addCondiments() {
        System.out.println("Adding sugar and milk");
    }
}

class Tea extends Beverage {
    void brew() {
        System.out.println("Steeping tea");
    }

    void addCondiments() {
        System.out.println("Adding lemon");
    }
}

public class TemplatePatternExample {
    public static void main(String[] args) {
```

```
            Beverage coffee = new Coffee();
            coffee.prepare();

            System.out.println();

            Beverage tea = new Tea();
            tea.prepare();
        }
    }
```

Références & sources

- Design Patterns in Java
- TutorialsPoint
- OpenClassrooms

Last updated on June 18, 2023

< DevOps

Centralisation des données >