**TYPE DE DESIGN PATTERNS**

Les design patterns, ou motifs de conception en français, sont des solutions réutilisables à des problèmes courants rencontrés dans le domaine de la conception logicielle. Ils fournissent des solutions éprouvées et élégantes pour des problèmes récurrents, facilitant ainsi le développement de logiciels robustes et maintenables. Voici quelques types de design patterns courants, classés en catégories :

1. **Créationnel**
2. **Singleton**

Le design pattern Singleton est un modèle de conception créationnel qui garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance. Il est souvent utilisé lorsque vous avez besoin d'une seule instance d'une classe pour coordonner des actions dans l'ensemble du système, comme dans le cas de la gestion des configurations, des connexions à une base de données, ou d'autres ressources partagées.

Exemple d’implémentation en java :

public class Singleton {

private static Singleton instance;

private Singleton() {}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

1. **Factory Method**

Le design pattern Factory Method est un modèle de conception créationnel qui fournit une interface pour créer des objets dans une classe, mais permet aux sous-classes de modifier le type d'objets qui seront créés. Il encapsule la création d'objets et délègue cette responsabilité aux classes dérivées.

**Exemple d'implémentation en Java :**

public interface LoggerFactory {

Logger createLogger();

}

public class ConsoleLoggerFactory implements LoggerFactory {

@Override

public Logger createLogger() {

return new ConsoleLogger();

}

}

public class FileLoggerFactory implements LoggerFactory {

@Override

public Logger createLogger() {

return new FileLogger();

}

}

public interface Logger {

void logMessage(String message);

}

public class ConsoleLogger implements Logger {

@Override

public void logMessage(String message) {

System.out.println("Console Logger: " + message);

}

}

public class FileLogger implements Logger {

@Override

public void logMessage(String message) {

System.out.println("File Logger: " + message);

}

}

1. **Abstract Factory**

Fournit une interface pour créer des familles d'objets apparentés ou dépendants sans spécifier leurs classes concrètes.

Exemple d’implémentation en java :

interface AbstractFactory {

ProductA createProductA();

ProductB createProductB();

}

interface ProductA {

void create();

}

interface ProductB {

void create();

}

class ConcreteFactory implements AbstractFactory {

@Override

public ProductA createProductA() {

return new ConcreteProductA();

}

@Override

public ProductB createProductB() {

return new ConcreteProductB();

}

}

class ConcreteProductA implements ProductA {

@Override

public void create() {

System.out.println("Product A created");

}

}

1. **Builder**

Permet de construire un objet complexe étape par étape. Utile lorsque la construction d'un objet directement en une seule étape est compliquée.

Example :

class Product {

private String part1;

private String part2;

public void setPart1(String part1) {

this.part1 = part1;

}

public void setPart2(String part2) {

this.part2 = part2;

}

@Override

public String toString() {

return "Product{part1='" + part1 + "', part2='" + part2 + "'}";

}

}

interface Builder {

void buildPart1();

void buildPart2();

Product getResult();

}

class ConcreteBuilder implements Builder {

private Product product = new Product();

@Override

public void buildPart1() {

product.setPart1("Part1");

}

@Override

public void buildPart2() {

product.setPart2("Part2");

}

@Override

public Product getResult() {

return product;

}

}

class Director {

public void construct(Builder builder) {

builder.buildPart1();

builder.buildPart2();

}

}

1. **Prototype**

Crée de nouveaux objets en copiant un objet existant, appelé le prototype. Cela peut être plus efficace que de créer de nouveaux objets à partir de zéro.

Example:

class Prototype implements Cloneable {

@Override

protected Object clone () throws CloneNotSupportedException {

return super. Clone ();

}

}

1. **Structural**
2. **Adapter**

Permet à une interface existante d'être utilisée comme une autre interface.

Exemple :

interface Target {

void request();

}

class Adaptee {

public void specificRequest() {

System.out.println("Specific Request");

}

}

class Adapter implements Target {

private Adaptee adaptee;

public Adapter(Adaptee adaptee) {

this.adaptee = adaptee;

}

@Override

public void request() {

adaptee.specificRequest();

}

}

1. **Bridge**

Sépare une abstraction de son implémentation de manière à ce que les deux puissent varier indépendamment.

Exemple :

interface Implementor {

void operationImp();

}

abstract class Abstraction {

protected Implementor implementor;

public Abstraction(Implementor implementor) {

this.implementor = implementor;

}

abstract void operation();

}

class ConcreteImplementorA implements Implementor {

@Override

public void operationImp() {

System.out.println("Concrete Implementor A");

}

}

class ConcreteImplementorB implements Implementor {

@Override

public void operationImp() {

System.out.println("Concrete Implementor B");

}

}

class RefinedAbstraction extends Abstraction {

public RefinedAbstraction(Implementor implementor) {

super(implementor);

}

@Override

void operation() {

implementor.operationImp();

}

}

1. **Composite**

Permet de traiter les objets individuels et les compositions d'objets de manière uniforme.

Exemple :

import java.util.ArrayList;

import java.util.List;

interface Component {

void operation();

}

class Leaf implements Component {

@Override

public void operation() {

System.out.println("Leaf Operation");

}

}

class Composite implements Component {

private List<Component> children = new ArrayList<>();

public void add(Component component) {

children.add(component);

}

@Override

public void operation() {

System.out.println("Composite Operation");

for (Component child : children) {

child.operation();

}

}

}

1. **Decorator**

Attache dynamiquement des responsabilités supplémentaires à un objet.

Exemple :

interface Component {

void operation();

}

class ConcreteComponent implements Component {

@Override

public void operation() {

System.out.println("Concrete Component Operation");

}

}

abstract class Decorator implements Component {

protected Component component;

public Decorator(Component component) {

this.component = component;

}

@Override

public void operation() {

component.operation();

}

}

class ConcreteDecoratorA extends Decorator {

public ConcreteDecoratorA(Component component) {

super(component);

}

@Override

public void operation() {

super.operation();

System.out.println("Concrete Decorator A Operation");

}

}

class ConcreteDecoratorB extends Decorator {

public ConcreteDecoratorB(Component component) {

super(component);

}

@Override

public void operation() {

super.operation();

System.out.println("Concrete Decorator B Operation");

}

}

1. **Facade**

Fournit une interface unifiée à un ensemble d'interfaces dans un sous-système. Simplifie l'utilisation d'un ensemble complexe de classes.

Exemple :

class SubsystemA {

public void operationA() {

System.out.println("Subsystem A Operation");

}

}

class SubsystemB {

public void operationB() {

System.out.println("Subsystem B Operation");

}

}

class Facade {

private SubsystemA subsystemA;

private SubsystemB subsystemB;

public Facade() {

this.subsystemA = new SubsystemA();

this.subsystemB = new SubsystemB();

}

public void operation() {

subsystemA.operationA();

subsystemB.operationB();

}

}

1. **Flyweight**

Utilise le partage pour prendre en charge efficacement un grand nombre d'objets qui ont des parties identiques.

Exemple :

import java.util.HashMap;

import java.util.Map;

interface Fly

1. **Proxy**

Fournit un substitut ou un espace réservé pour un autre objet pour contrôler l'accès à cet objet.

Exemple :

interface RealSubject {

void request();

}

class RealSubjectImpl implements RealSubject {

@Override

public void request() {

System.out.println("RealSubject: Traitement de la requête");

}

}

class Proxy implements RealSubject {

private RealSubjectImpl realSubject;

@Override

public void request() {

if (realSubject == null) {

realSubject = new RealSubjectImpl();

}

realSubject.request();

}

}

public class ProxyExample {

public static void main(String[] args) {

RealSubject proxy = new Proxy();

proxy.request();

}

}

1. **Comportemental**
2. **Chain of Responsibility**

Permet à plusieurs objets de gérer une requête sans que l'émetteur de la requête ne connaisse les objets qui la traitent.

Exemple :

public abstract class Approver {

protected Approver successor;

public void setSuccessor(Approver successor) {

this.successor = successor;

}

public abstract void processRequest(PurchaseRequest request);

}

public class Manager extends Approver {

private static final double MAX\_LIMIT = 10000.0;

@Override

public void processRequest(PurchaseRequest request) {

if (request.getAmount() <= MAX\_LIMIT) {

System.out.println("Manager approves purchase request #" + request.getRequestNumber());

} else if (successor != null) {

successor.processRequest(request);

}

}

}

// ConcreteHandler (Gestionnaire Concret)

public class Director extends Approver {

private static final double MAX\_LIMIT = 25000.0;

@Override

public void processRequest(PurchaseRequest request) {

if (request.getAmount() <= MAX\_LIMIT) {

System.out.println("Director approves purchase request #" + request.getRequestNumber());

} else if (successor != null) {

successor.processRequest(request);

}

}

}

public class VicePresident extends Approver {

private static final double MAX\_LIMIT = 50000.0;

@Override

public void processRequest(PurchaseRequest request) {

if (request.getAmount() <= MAX\_LIMIT) {

System.out.println("Vice President approves purchase request #" + request.getRequestNumber());

} else if (successor != null) {

successor.processRequest(request);

}

}

}

public class President extends Approver {

private static final double MAX\_LIMIT = 100000.0;

@Override

public void processRequest(PurchaseRequest request) {

if (request.getAmount() <= MAX\_LIMIT) {

System.out.println("President approves purchase request #" + request.getRequestNumber());

} else {

System.out.println("Purchase request #" + request.getRequestNumber() + " requires board meeting");

}

}

}

// Request (Requête)

public class PurchaseRequest {

private int requestNumber;

private double amount;

public PurchaseRequest(int requestNumber, double amount) {

this.requestNumber = requestNumber;

this.amount = amount;

}

public int getRequestNumber() {

return requestNumber;

}

public double getAmount() {

return amount;

}

}

// Client

public class Client {

public static void main(String[] args) {

Approver manager = new Manager();

Approver director = new Director();

Approver vp = new VicePresident();

Approver president = new President();

manager.setSuccessor(director);

director.setSuccessor(vp);

vp.setSuccessor(president);

PurchaseRequest request1 = new PurchaseRequest(1, 8000.0);

PurchaseRequest request2 = new PurchaseRequest(2, 25000.0);

PurchaseRequest request3 = new PurchaseRequest(3, 60000.0);

PurchaseRequest request4 = new PurchaseRequest(4, 120000.0);

manager.processRequest(request1);

manager.processRequest(request2);

manager.processRequest(request3);

manager.processRequest(request4);

}

}

1. **Command**

Encapsule une requête en tant qu'objet, permettant ainsi de paramétrer les clients avec différentes requêtes, files d'attente et journaux de requêtes. Ce pattern encapsule une requête en tant qu'objet, permettant ainsi de paramétrer des clients avec différentes requêtes, de mettre en file d'attente ou de journaliser des requêtes et de supporter des opérations annulables. Par exemple, dans une application de traitement de texte, les commandes telles que "copier", "coller" et "annuler" peuvent être implémentées à l'aide du pattern Command.

Exemple :

interface Command {

void execute();

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

)

class Light {

public void turnOn() {

System.out.println("Light is ON");

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

1. **Interpreter**

Donne une représentation grammaticale d'un langage et un interpréteur pour interpréter les phrases de ce langage.

Exemple :

class Select implements Expression {

private String column;

private From from;

@Override

public List<String> interpret(Context ctx) {

ctx.setColumn(column);

return from.interpret(ctx);

}

}

1. **Iterator**

Fournit un moyen de parcourir séquentiellement les éléments d'une collection sans exposer la représentation interne de la collection. Ce pattern fournit un moyen d'accéder séquentiellement aux éléments d'une collection sans exposer sa structure interne. Il permet de parcourir une collection d'objets sans connaître les détails de sa mise en œuvre. Par exemple, dans une application de gestion de fichiers, un itérateur peut être utilisé pour parcourir les fichiers d'un répertoire

Exemple :

1. **Mediator**

Définit un objet qui encapsule la manière dont un ensemble d'objets interagit. Favorise un couplage lâche en évitant que les objets ne se réfèrent les uns aux autres explicitement.

Exemple :

public class NameRepository implements Container {

public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

@Override

public Iterator getIterator() {

return new NameIterator();

}

private class NameIterator implements Iterator {

int index;

@Override

public boolean hasNext() {

if(index < names.length){

return true;

}

return false;

}

@Override

public Object next() {

if(this.hasNext()){

return names[index++];

}

return null;

}

}

}

1. **Memento**

Permet de capturer et d'externaliser un état interne d'un objet, de manière à pouvoir restaurer l'objet à cet état ultérieurement.

Exemple :

public class TextWindowState {

private String text;

public TextWindowState(String text) {

this.text = text;

}

public String getText() {

return text;

}

}

1. **Observer**

Définit une dépendance entre objets de manière à ce que lorsque l'état d'un objet change, tous ses dépendants soient notifiés et mis à jour automatiquement. Ce pattern permet de définir une famille d'algorithmes interchangeables et de les encapsuler individuellement, de sorte qu'ils puissent être utilisés de manière interchangeable. Par exemple, dans une application de traitement d'images, différents algorithmes de compression peuvent être utilisés en fonction des préférences de l'utilisateur.

Exemple :

import java.util.ArrayList;

import java.util.List;

class Subject {

private List<Observer> observers = new ArrayList<>();

public void addObserver(Observer observer) {

observers.add(observer);

}

public void removeObserver(Observer observer) {

observers.remove(observer);

}

public void notifyObservers() {

for (Observer observer : observers) {

observer.update();

}

}

}

interface Observer {

void update();

}

class ConcreteObserver implements Observer {

@Override

public void update() {

System.out.println("Notification reçue");

}

}

1. **State**

Permet à un objet de modifier son comportement lorsqu'il change son état interne.

Exemple :

package com.journaldev.design.state;

public interface State {

public void doAction();

}

package com.journaldev.design.state;

public class TVStopState implements State {

@Override

public void doAction() {

System.out.println("TV is turned OFF");

}

}

1. **Strategy**

Définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables.

Exemple :

interface SortingStrategy {

void sort(int[] array);

}

class BubbleSortStrategy implements SortingStrategy {

@Override

public void sort(int[] array) {

System.out.println("Tri à bulles");

}

}

class QuickSortStrategy implements SortingStrategy {

@Override

public void sort(int[] array) {

// Implémentation du tri rapide

System.out.println("Tri rapide");

}

}

class SorterContext {

private SortingStrategy strategy;

public void setStrategy(SortingStrategy strategy) {

this.strategy = strategy;

}

public void performSort(int[] array) {

strategy.sort(array);

}

}

1. **Template Method**

Définit le squelette d'un algorithme dans une méthode, mais permet aux sous-classes de redéfinir certaines étapes de l'algorithme sans changer sa structure. Ce pattern définit le squelette d'un algorithme dans une méthode, en laissant certaines étapes aux sous-classes pour l'implémentation. Il permet de définir une structure générale d'un algorithme tout en permettant aux sous-classes de fournir des implémentations spécifiques. Par exemple, dans un jeu vidéo, différentes classes de personnages peuvent hériter d'une classe de base qui définit une méthode de combat, tandis que chaque classe spécifique de personnage peut fournir sa propre implémentation pour cette méthode.

Exemple :

package com.journaldev.design.template;

public abstract class HouseTemplate {

//template method, final so subclasses can't override

public final void buildHouse(){

buildFoundation();

buildPillars();

buildWalls();

buildWindows();

System.out.println("House is built.");

}

//default implementation

private void buildWindows() {

System.out.println("Building Glass Windows");

}

//methods to be implemented by subclasses

public abstract void buildWalls();

public abstract void buildPillars();

private void buildFoundation() {

System.out.println("Building foundation with cement,iron rods and sand");

}

}

1. **Visitor**

Représente une opération à effectuer sur les éléments d'une structure d'objets. Permet de définir une nouvelle opération sans changer les classes des éléments sur lesquels elle opère.

Exemple :

public class ElementVisitor implements Visitor {

@Override

public void visit(XmlElement xe) {

System.out.println(

"processing an XML element with uuid: " + xe.uuid);

}

@Override

public void visit(JsonElement je) {

System.out.println(

"processing a JSON element with uuid: " + je.uuid);

}

}