Tervezési minták egy OO programozási nyelvben

1.Model-View-Controller (MVC)

Az MVC (Model-View-Controller) minta célja az alkalmazások logikai rétegeinek elkülönítése. Ezáltal az adatok (Model), a megjelenítés (View) és a vezérlés (Controller) külön fejleszthetők és karbantarthatók.

Előnyök:

- - Elkülöníti a logikát a megjelenítéstől.
- - Könnyebb karbantarthatóság és tesztelhetőség.
- - Többféle megjelenítés ugyanazokkal az adatokkal.

Hátrányok:

- - Bonyolultabb alkalmazások esetén megnő a komplexitás.
- - Több fájlt és osztályt kell kezelni.

Implementáció:

A Model felelős az adatokért és logikáért, a View a megjelenítésért, a Controller pedig az adatfolyamot irányítja közöttük.

Java Példakód:

```
public class MVCExample {
public static void main(String[] args) {
Model model = new Model("Hello MVC");
View view = new View();
Controller controller = new Controller(model, view);
controller.updateView();
}
}

class Model {
private String data;
public Model(String data) { this.data = data; }
public String getData() { return data; }
public void setData(String data) { this.data = data; }
}

class View {
public void printData(String data) {
System.out.println("Data: " + data);
}
}

class Controller {
private Model model;
private View view;
public Controller(Model model, View view) {
this.model = model;
this.view = view;
}
public void updateView() {
view.printData(model.getData());
}
}
```

2.Singleton

A Singleton minta biztosítja, hogy egy osztályból csak egyetlen példány létezhessen, és azt egy globális hozzáférési ponton keresztül érjük el.

Előnyök:

- - Biztosítja, hogy csak egy példány létezzen.
- - Könnyen hozzáférhető globális objektum.

Hátrányok:

- - Nehezebb tesztelni, mivel az állapotot megosztja a kód különböző részei között.
- - Túlzott használata anti-pattern lehet.

Implementáció:

Privát konstruktort használ az objektum létrehozásának korlátozására, és egy statikus metódust biztosít az egyetlen példány eléréséhez.

Java Példakód:

```
public class SingletonExample {
private static SingletonExample instance;
private SingletonExample() {}
public static SingletonExample getInstance() {
```

```
if (instance == null) {
instance = new SingletonExample();
}
return instance;
}
}
```

Singleton

A Singleton minta biztosítja, hogy egy osztályból csak egy példány létezzen az alkalmazás futása során, és azt egy globális hozzáférési ponton keresztül érjük el.

Implementáció:

Privát konstruktorral akadályozzuk meg az osztály példányosítását kívülről, és statikus metódust biztosítunk a példány eléréséhez.

Java Példakód:

```
public class Singleton {
private static Singleton instance;
private Singleton() {}
public static Singleton getInstance() {
if (instance == null) {
instance = new Singleton();
}
return instance;
}
```

```
}
```

3.Factory Method

A Factory Method minta lehetővé teszi, hogy egy osztály meghatározza az objektumok létrehozásának alapját, de az alosztályok döntik el, milyen típusú objektumokat hoznak létre.

Implementáció:

Egy metódust definiálunk, amely egy interfész példányát hozza létre az alosztályok implementációi alapján.

Java Példakód:

```
interface Product {
void create();
}

class ConcreteProduct implements Product {
public void create() {
System.out.println("ConcreteProduct created");
}
}

class ProductFactory {
public static Product createProduct() {
return new ConcreteProduct();
}
```

```
}

public class FactoryExample {
public static void main(String[] args) {
Product product = ProductFactory.createProduct();
product.create();
}
}
```

4.Adapter

Az Adapter minta lehetővé teszi, hogy egy osztály illeszkedjen egy másik osztály interfészéhez, amit az ügyfél elvár, így az osztályok együtt tudnak működni.

Implementáció:

Az Adapter osztály becsomagolja az eredeti osztályt, és az elvárt interfészt biztosítja az ügyfél számára.

Java Példakód:

```
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;
}
public void request() {
adaptee.specificRequest();
}
}

public class AdapterExample {
public static void main(String[] args) {
Adaptee adaptee = new Adaptee();
Target target = new Adapter(adaptee);
target.request();
}
}
```

5.Command

A Command minta lehetővé teszi, hogy a műveleteket parancsobjektumokba csomagoljuk, amelyek végrehajthatók, visszavonhatók vagy naplózhatók.

Implementáció:

A Command interfészt implementáló osztályok definiálják a végrehajtandó műveleteket, amelyeket a meghívó objektum hív meg.

Java Példakód:

```
interface Command {
void execute();
}

class Light {
public void turnOn() {
System.out.println("The light is ON");
}
}

class TurnOnCommand implements Command {
private Light light;
public TurnOnCommand(Light light) {
this.light = light;
}
public void execute() {
light.turnOn();
}
}

class RemoteControl {
private Command command;
public void setCommand(Command command) {
this.command = command;
}
public void pressButton() {
command.execute();
}
}
```

```
public class CommandExample {
public static void main(String[] args) {
Light light = new Light();
Command turnOn = new TurnOnCommand(light);
RemoteControl remote = new RemoteControl();
remote.setCommand(turnOn);
remote.pressButton();
}
}
```

6.Observer

Az Observer minta lehetővé teszi, hogy egy objektum állapotváltozásait több másik objektum kövesse, és automatikusan értesítést kapjon ezekről.

Implementáció:

Az alany (Subject) osztály tartja nyilván a megfigyelőket, és értesíti őket, ha változás történik.

Java Példakód:

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

interface Observer {
void update(String message);
```

```
}

class Subject {
private List<Observer> observers = new ArrayList<>();
public void addObserver(Observer observer) {
observers.add(observer);
}
public void notifyObservers(String message) {
for (Observer observer : observers) {
observer.update(message);
}
}
}

class ConcreteObserver implements Observer {
private String name;
public ConcreteObserver(String name) {
this.name = name;
}
public void update(String message) {
System.out.println(name + " received: " + message);
}
}

public class ObserverExample {
public static void main(String[] args) {
Subject subject = new Subject();
Observer observer1 = new ConcreteObserver("Observer 1");
Observer observer2 = new ConcreteObserver("Observer 2");
subject.addObserver(observer1);
subject.addObserver(observer2);
subject.notifyObservers("Hello Observers");
}
```

```
}
```