TERVEZÉSI MINTÁK OO PROGRAMOZÁSI NYELVBEN

Az informatikában a programtervezési mintának (angolul Software Design Patterns) nevezik a gyakran előforduló programozási feladatokra adható általános, újra felhasználható megoldásokat. Egy programtervezési minta rendszerint egymással együttműködő objektumok és osztályok leírása

A tervminták nem nyújtanak kész tervet, amit közvetlenül le lehet kódolni, habár vannak hozzájuk példakódok, amiket azonban meg kell tölteni az adott helyzetre alkalmas kóddal. Céljuk az, hogy leírást vagy sablont nyújtsanak. Segítik formalizálni a megoldást.

A minták rendszerint osztályok és objektumok közötti kapcsolatokat mutatnak, de nem specifikálják konkrétan a végleges osztályokat vagy objektumokat. A modellek absztrakt osztályai helyett egyes esetekben interfészek is használhatók, habár azokat maga a tervminta nem mutatja. Egyes nyelvek beépítetten tartalmaznak tervmintákat. A tervminták tekinthetők a strukturált programozás egyik szintjének a paradigma és az algoritmus között.

A tervezési mintákat **3 fő** kategóriába sorolhatjuk:

- *létrehozási minta*
- *szerkezeti minta*
- *viselkedési minta*

Készítette:	***Éles Máté***
Készítés dátuma:	***2024.12.03.***

1. Létrehozási Minta (Creational Pattern)

A létrehozási minták célja az objektumok létrehozási folyamatának kezelése. Ezek a minták elválasztják a konkrét osztály példányosítását az osztály felhasználásától, így a kód rugalmasabbá válik az objektumok létrehozásában. Az egyik legnépszerűbb létrehozási minta a Builder minta.

Példa: Builder minta

A Builder minta lehetővé teszi komplex objektumok lépésenkénti felépítését. Ahelyett, hogy a konstruktort használva egyszerre adnánk meg minden szükséges paramétert, a Builder mintával fokozatosan építjük fel az objektumot.

hello-world.js

```
public class House {
    private String foundation;
    private String structure;
    private String roof;
    private boolean hasGarden;
    private boolean hasGarage;

    // Private constructor, only the Builder can instantiate
    private House(HouseBuilder builder) {
        this.foundation = builder.foundation;
        this.structure = builder.structure;
        this.roof = builder.roof;
        this.hasGarden = builder.hasGarden;
        this.hasGarage = builder.hasGarage;
    }

    public static class HouseBuilder {
        private String foundation;
        private String structure;
        private String roof;
        private boolean hasGarden;
        private boolean hasGarage;

        public HouseBuilder(String foundation, String structure, String roof)
{
            this.foundation = foundation;
            this.structure = structure;
            this.roof = roof;
        }

        public HouseBuilder setGarden(boolean hasGarden) {
            this.hasGarden = hasGarden;
            return this;
        }

        public HouseBuilder setGarage(boolean hasGarage) {
            this.hasGarage = hasGarage;
            return this;
        }

        public House build() {
            return new House(this);
        }
    }

    @Override
    public String toString() {
        return "House with " + foundation + " foundation, " + structure + "
structure, " + roof + " roof" +
                (hasGarden ? ", with garden" : "") + (hasGarage ? ", with
garage" : "");
    }
}

// Használat:
House house = new House.HouseBuilder("Concrete", "Brick", "Tile")
    .setGarden(true)
    .setGarage(true)
    .build();
System.out.println(house);
```

Ez a kód egy ház objektumot épít lépésenként, ami rugalmasabb és olvashatóbb kódot eredményez, különösen, ha sok beállítási paraméter van.

2. Szerkezeti minta (Structural Pattern)

A szerkezeti minták az objektumok közötti kapcsolatok megszervezésére és szerkezeti felépítésük hatékonyabb kezelésére összpontosítanak. Ezek a minták segítenek különböző objektumok összekapcsolásában és integrálásában, gyakran anélkül, hogy a kód szorosan kapcsolódna egymáshoz. Az egyik ilyen minta a Decorator minta.

Példa: Decorator minta

A Decorator minta lehetővé teszi új funkcionalitások dinamikus hozzáadását egy objektumhoz anélkül, hogy módosítanánk az eredeti osztály kódját.

hello-world.js

```
// Az alap interfész
public interface Coffee {
    String getDescription();
    double getCost();
}

// Egy konkrét Coffee implementáció
public class SimpleCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "Simple Coffee";
    }

    @Override
    public double getCost() {
        return 5.0;
    }
}

// A Decorator, amely egy másik Coffee-t díszít
public class MilkDecorator implements Coffee {
    private Coffee decoratedCoffee;

    public MilkDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", with Milk";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 1.5;
    }
}

public class SugarDecorator implements Coffee {
    private Coffee decoratedCoffee;

    public SugarDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", with Sugar";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.5;
    }
}

// Használat:
Coffee coffee = new SimpleCoffee();
System.out.println(coffee.getDescription() + " $" + coffee.getCost());

coffee = new MilkDecorator(coffee);
System.out.println(coffee.getDescription() + " $" + coffee.getCost());

coffee = new SugarDecorator(coffee);
System.out.println(coffee.getDescription() + " $" + coffee.getCost());
```

Ebben a példában különböző dekorátorokat adunk a kávéhoz, így dinamikusan bővítve az eredeti objektum funkcionalitását anélkül, hogy módosítanánk a SimpleCoffee *osztály kódját.*

3. Viselkedési minta (Behavioral Pattern)

A viselkedési minták az objektumok közötti kommunikációt és együttműködést optimalizálják. Az egyik ilyen minta a Strategy minta, amely lehetővé teszi különböző algoritmusok közötti választást a futásidő alatt, anélkül, hogy az algoritmusok felhasználását módosítani kellene.

Példa: Strategy minta

A Strategy minta különféle algoritmusokat definiál és különböző implementációk között váltás lehetőségét biztosítja anélkül, hogy a kódot módosítani kellene.

hello-world.js

```
// A stratégia interfész
public interface PaymentStrategy {
    void pay(int amount);
}

// Két konkrét stratégia
public class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;

    public CreditCardPayment(String cardNumber) {
        this.cardNumber = cardNumber;
    }

    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Credit Card: " +
cardNumber);
    }
}

public class PayPalPayment implements PaymentStrategy {
    private String email;

    public PayPalPayment(String email) {
        this.email = email;
    }

    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using PayPal: " + email);
    }
}

// A context osztály, amely a stratégiai mintát alkalmazza
public class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void checkout(int amount) {
        paymentStrategy.pay(amount);
    }
}

// Használat:
ShoppingCart cart = new ShoppingCart();
cart.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456"));
cart.checkout(100);  // Output: Paid 100 using Credit Card: 1234-5678-9012-
3456

cart.setPaymentStrategy(new PayPalPayment("user@example.com"));
cart.checkout(200);  // Output: Paid 200 using PayPal: user@example.com
```

Ebben a példában a fizetési stratégiát dinamikusan változtathatjuk anélkül, hogy a ShoppingCart *osztály logikáját meg kellene változtatnunk.*