Tervezési minták egy OO programozási nyelvben. MVC, mint modell-nézet-vezérlő minta és néhány másik tervezési minta

MVC (Model-View-Controller), mint modell-nézet-vezérlő minta

Az MVC (Model-View-Controller) **egy szerkezeti tervezési minta, amely a szoftveres alkalmazásokat három, egymástól logikailag elkülönülő, de együttműködő részre osztja a felhasználói felület (GUI) és az üzleti logika szétválasztása érdekében.** Ez a minta különösen népszerű a webes és asztali alkalmazások például a Java Swing vagy a JavaFX alapú GUI-k fejlesztésénél.

Az MVC komponensei

Az MVC minta három fő részből áll: a Modellből (Model), a Nézetből (View) és a Vezérlőből (Controller). Ezek a komponensek együttműködve biztosítják, hogy az alkalmazás logikája, megjelenítése és a felhasználói interakciók jól elkülönüljenek.

- **Model (Modell):** A Model az alkalmazás adatait és üzleti logikáját kezeli. **Feladata az adatok tárolása, feldolgozása és a szükséges számítások elvégzése.** Nem tartalmaz semmilyen grafikus elemet vagy megjelenítési logikát. Például egy olyan osztály, amely az összeadást vagy az adatmentést végzi.
- **View (Nézet): A View felel az adatok felhasználó számára történő megjelenítéséért. Ez tartalmazza a grafikus elemeket, például ablakokat, gombokat és szövegmezőket.** A View maga nem végez logikai műveleteket; csupán megjeleníti az adatokat, és a felhasználói eseményeket továbbítja a vezérlőnek.
- **Controller (Vezérlő): A Controller teremti meg a kapcsolatot a Model és a View között.** Kezeli a felhasználói interakciókat, meghívja a Model megfelelő metódusait, majd az eredményt visszaküldi a View-nak megjelenítésre. Így a vezérlő biztosítja az adatok áramlását az egyes rétegek között.

Egyszerű Java GUI példa: Két szám összeadása

1. Model (Modell)

A Model az alkalmazás logikai rétege. Ez az osztály felelős az adatok kezeléséért és az üzleti logika végrehajtásáért. A példa esetében ez csupán két szám összeadását végzi el.

```java
package org.example;

public class Model {
    // két szám összeadása
    public int osszead(int a, int b) {
        return a + b;
    }
}
```

2. View (Nézet)

A View osztály felel a grafikus felület megjelenítéséért. A Swing könyvtárat használja, és tartalmaz bemeneti mezőket, egy gombot és egy eredménymezőt. A View nem végez számításokat — csak megjelenít és jelzéseket küld a Controllernek.

```java
package org.example;

import javax.swing.*;
import java.awt.*;

public class View extends JFrame {
    private JTextField elsoSzam = new JTextField(10);
    private JTextField masodikSzam = new JTextField(10);
    private JButton osszeadGomb = new JButton("Összead");
    private JTextField eredmenyMezo = new JTextField(10);

    public View() {
        super("Egyszerű MVC kalkulátor");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 150);
        setLayout(new GridLayout(4, 2, 5, 5));

        add(new JLabel("Első szám:"));
        add(elsoSzam);
        add(new JLabel("Második szám:"));
        add(masodikSzam);
        add(osszeadGomb);
        add(new JLabel("Eredmény:"));
        add(eredmenyMezo);

        eredmenyMezo.setEditable(false);
        setVisible(true);
    }

    public int getElsoSzam() { return Integer.parseInt(elsoSzam.getText()); }
    public int getMasodikSzam() { return Integer.parseInt(masodikSzam.getText()); }
    public void setEredmeny(int eredmeny) { eredmenyMezo.setText(String.valueOf(eredmeny)); }
    public JButton getOsszeadGomb() { return o eadGomb; }
}
```

3. Controller (Vezérlő)

A Controller köti össze a View és a Model rétegeket. Figyeli a felhasználói eseményeket (pl. gombnyomás), lekéri a View-ból az adatokat, meghívja a Model logikáját, és az eredményt visszaadja a View-nak.

```java
package org.example;

import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Controller {
    private final View view;
    private final Model model;

    public Controller(View view, Model model) {
        this.view = view;
        this.model = model;
        this.view.getOsszeadGomb().addActionListener(new OsszeadListener());
    }

    private class OsszeadListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            try {
                int szam1 = view.getElsoSzam();
                int szam2 = view.getMasodikSzam();
                int eredmeny = model.osszead(szam1, szam2);
                view.setEredmeny(eredmeny);
            } catch (NumberFormatException ex) {
                JOptionPane.showMessageDialog(view, "Kérlek, számokat adj meg!");
            }
        }
    }
}
```

4. Main (Program indítása)

A főprogram (Main osztály) példányosítja és összekapcsolja az MVC komponenseket. Itt jön létre a View, a Model és a Controller, így a program működőképes egésszé válik.

```java
package org.example;

import javax.swing.SwingUtilities;

public class Main {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            View view = new View();
            Model model = new Model();
            new Controller(view, model);
        });
    }
}
```

Hogyan működik az MVC minta ebben a példában?

1. A felhasználó beír két számot és megnyomja az „Összead” gombot. (View)
2. A gombnyomást az ActionListener érzékeli. (Controller)
3. A Controller lekéri a View-ból a két számot. (Controller → View)
4. A Controller meghívja a Model osszead() metódusát. (Controller → Model)
5. A Model visszaadja az eredményt. (Model → Controller)
6. A Controller átadja az eredményt a View-nak, ami kiírja a mezőbe. (Controller → View)

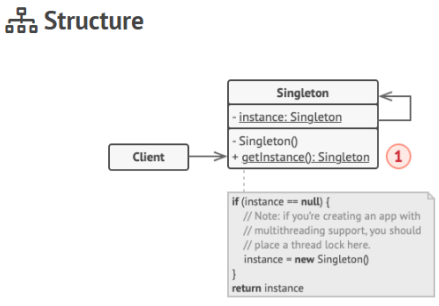
Java tervezési minták – Singleton minta

A Java tervezési minták strukturált, újrahasznosítható és karbantartható kód írását segítik. Három fő kategóriájuk van: létrehozási (creational), szerkezeti (structural) és viselkedési (behavioral) minták. A létrehozási minták az objektumok létrehozásának módját szabályozzák, egyik legismertebb példájuk a **Singleton minta**.
A **Singleton minta** célja, hogy **egy osztályból csak egyetlen példány** (objektum) jöhessen létre a program teljes futása alatt, és ehhez **globális hozzáférési pontot** biztosítson.

Előnyei:

- Garantálja az egyetlen példány létrejöttét.
- Központi hozzáférést biztosít a példányhoz.
- Elkerüli az erőforrás-pazarlást.
- Egyszerűen megvalósítható és használható.

Megvalósítás lépései:



1. **Attribútum: - instance: Singleton**

Ez egy **privát statikus változó**, amely a **Singleton egyetlen példányát** tárolja.

A - jel a **privát** láthatóságot jelenti (csak az osztályon belül érhető el).

Az ábrán az instance: Singleton mezőből a Singleton-ra mutató nyíl azt, jelenti, hogy ez a mező a Singleton osztály egy példányára (objektumára) mutat, vagyis önmagát tárolja, egy statikus referencia formájában.

2. **Konstruktor: - Singleton()**

A konstruktor is **privát**, hogy **ne lehessen kívülről új példányt létrehozni** (new Singleton() tiltott a kliens számára). Eltárolja az egyetlen példányt.

3. **Statikus metódus: + getInstance(): Singleton**

Ez egy **publikus statikus metódus**, amely a Singleton példányát adja vissza és szükség esetén létrehozza azt.

A + jel a **publikus** láthatóságot jelenti.

A működés logikája

- **Ellenőrzi**, hogy a Singleton példány már létezik-e.
- Ha **nem létezik**, akkor **létrehozza** (new Singleton()).
- Ha már **létezik**, akkor **visszaadja** a korábbit.
- A „Client” nyíl azt mutatja, hogy a kliens **nem közvetlenül példányosítja** a Singleton-t (new kulcsszóval), hanem: Singleton s = Singleton.getInstance();
  Így a kliens mindig **a központilag kezelt egyetlen példányhoz** jut hozzá.

Singleton minta gyakorlati példa:

```
public class Singleton {
    private static Singleton instance;

    private Singleton() { }

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

Kód magyarázata:

- Ha **még nincs létrehozva** a Singleton példány (instance == null), akkor **létrehozza**.
- Ha **már létezik**, akkor **ugyanazt a példányt** adja vissza újra.

Példa a működésre:

```
public class Main {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        System.out.println(s1 == s2); // true → ugyanarra az objektumra mutat
    }
}
```

Mivel s1 és s2 ugyanarra az objektumra mutatnak, az eredmény true.

Összefoglalás

- Csak egyetlen példány jön létre a Singleton osztályból.
- A getInstance() mindig ugyanazt az objektumot adja vissza.
- Ezért a s1 és s2 változók nem két külön objektumot, hanem ugyanazt az egyet használják.

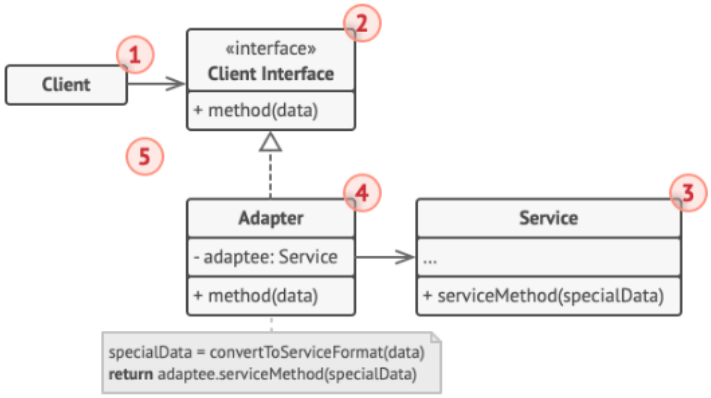
Java szerkezeti minták – Strukturális tervezési minta: Adapter minta

Az Adapter minta arra szolgál, hogy két egymással inkompatibilis osztályt össze lehessen kapcsolni. Egy „átalakító” osztályt hoz létre, amely közvetítőként működik két eltérő felület között.

Előnyei:

- **Régi kódot új kóddal lehet együtt használni.**
- Nem kell átírni egyik osztályt sem.
- Olyan, mint egy konnektor átalakító: más a csatlakozó, de mégis működik együtt.

Megvalósítás lépései:



1. Client (Kliens): Egy osztály, amely tartalmazza a program meglévő üzleti logikáját.
2. Client Interface (Kliens interfész): Egy protokoll, amelyet más osztályoknak követniük kell ahhoz, hogy együtt tudjanak működni a kliens kódjával.
3. Service (Szervíz): Egy meglévő, vagy régi (legacy) osztály. Hasznos funkciókat tartalmaz, de a kliens nem tudja közvetlenül használni, mert az interfésze nem kompatibilis.
4. Adapter: Az adapter hívásokat kap a klienstől a kliens interfészen keresztül, majd ezeket lefordítja a becsomagolt szerviz objektum számára érthető formátumra.
5. Mivel a kliens csak az interfészt használja és nem a konkrét adaptert, az adapterek szabadon cserélhetők vagy bővíthetők a klienskód módosítása nélkül, még akkor is, ha a szerviz interfésze megváltozik.

Java példa – Nyomtató adapter

- Van egy régi nyomtató osztály (OldPrinter), amit nem lehet módosítani.
- Az új kód egy Printer interfészt vár.
- Az Adapter összeköti a kettőt.

1. A kliens által elvárt interfész

```java
// A kliens ezt az interfészt ismeri
interface Printer {
    void print(String text);
}
```

2. A meglévő, „régi” nyomtató osztály (nem kompatibilis)

```java
// Régi osztály, más metódusnévvel
class OldPrinter {
    public void printText(String message) {
        System.out.println("OldPrinter nyomtat: " + message);
    }
}
```

3. Az Adapter – összeköti a kettőt

```java
// Az adapter megvalósítja az új Printer interfészt,
// és belül a régi OldPrintert használja
class PrinterAdapter implements Printer {

    private OldPrinter oldPrinter;

    public PrinterAdapter(OldPrinter oldPrinter) {
        this.oldPrinter = oldPrinter;
    }

    @Override
    public void print(String text) {
        // A hívás "lefordítása"
        oldPrinter.printText(text);
    }
}
```

4. A klienskód

```java
public class Client {
    public static void main(String[] args) {
        OldPrinter legacy = new OldPrinter();
        Printer printer = new PrinterAdapter(legacy); // adapter becsomagolja

        printer.print("Hello világ!");
    }
}
```

A kliens egy Printer interfészt szeretne használni.

A régi rendszerben csak egy **OldPrinter** van, amelynek más a metódusa.

A **PrinterAdapter** „lefordítja” a kliens hívásait az OldPrinter számára.

A kliensnek **nem kell tudnia** sem az adapterről, sem a régi eszközről — csak az interfészt használja.

Java szerkezeti minták – Observer (megfigyelő) minta

Az Observer minta egy viselkedési tervezési minta, amely azt teszi lehetővé, hogy egy objektum (a Subject) értesítést küldjön több másik objektumnak (az Observereknek) akkor, amikor az állapota megváltozik, anélkül, hogy szorosan össze lenne velük kötve.

Fő cél: Elválasztani egymástól az adatforrást és az arra reagáló komponenseket, így rugalmas, laza csatolású rendszert hoz létre.

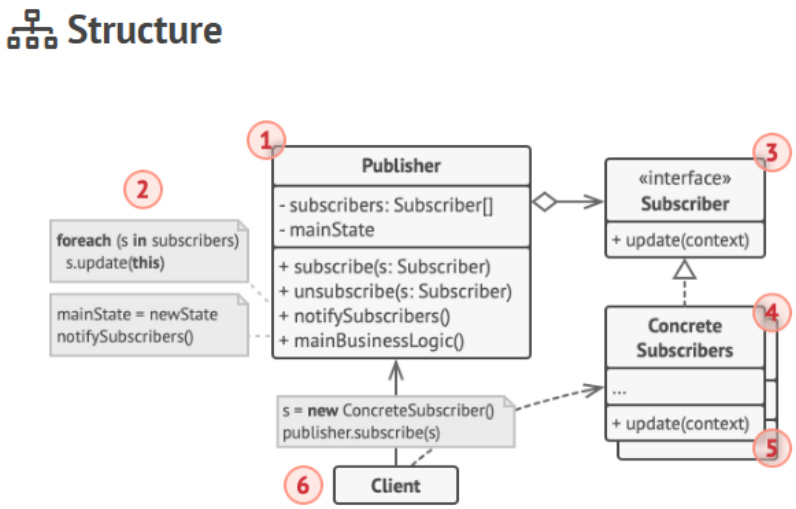
Előnyök:

- **Laza csatolás a Subject és az Observerek közt.**
- **Több komponens automatikusan frissülhet egyetlen adatváltozásra.**
- Könnyen bővíthető új Observerekkel.

Hátrányok:

- **Sok Observer esetén több értesítés → teljesítményromlás lehet.**
- A frissítések sorrendje nem garantált.

Megvalósítás lépései:



1. Publisher (Subject)

- Eseményeket bocsát ki, amikor megváltozik az állapota vagy valamilyen műveletet végrehajt.
- Fenntart egy feliratkozási listát (subscribers).
- Lehetővé teszi új feliratkozók hozzáadását és meglévők eltávolítását.
- Amikor esemény történik, végigmegy a listán és minden feliratkozón meghívja a notification (update) metódust.

2. Subscriber (Observer)

- Amikor új esemény következik be, a Publisher végigmegy a feliratkozók listáján, és meghívja a Subscriber interfészben deklarált értesítési metódust minden egyes Subscriber objektumon.

3. Subsriber

- A Subscriber interfész határozza meg az értesítési felületet. Ez általában egyetlen update metódusból áll, amely több paramétert is kaphat, hogy a Publisher át tudja adni az esemény részleteit.

4. Concrete Subscribers

- A konkrét Subscriber osztályok valamilyen műveletet hajtanak végre a Publisher értesítéseire reagálva. Minden Subscriber ugyanazt az interfészt valósítja meg, így a Publisher nincs konkrét osztályokhoz kötve.

5. Concrete Subscribers

- A Subscriber-ek gyakran igényelnek kontextusadatokat a frissítések helyes kezeléséhez. Emiatt a Publisher gyakran átad ilyen adatokat a notification (update) metódus paramétereként. Akár saját magát is átadhatja, hogy a Subscriber további információkat kérhessen le.

6. Client

- A Client külön hozza létre a Publisher és Subscriber objektumokat, majd feliratkoztatja a Subscriber-eket a Publisher értesítéseire.

Java Observer minta példa:

1. Publisher: kezeli a feliratkozók listáját, tárolja az állapotot és értesíti a Subscriber-eket.

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

public class WeatherStation implements Publisher {

    private List<Subscriber> subscribers = new ArrayList<>();
    private float temperature;   // mainState

    @Override
    public void subscribe(Subscriber s) {
        subscribers.add(s);
    }

    @Override
    public void unsubscribe(Subscriber s) {
        subscribers.remove(s);
    }

    @Override
    public void notifySubscribers() {
        for (Subscriber s : subscribers) {   // 2. pont
            s.update(this);                  // context átadás
        }
    }
```

```
    // Példa fő logika – állapotváltozás
    public void setTemperature(float newTemp) {
        this.temperature = newTemp;
        notifySubscribers();
    }

    public float getTemperature() {
        return temperature;
    }
}
```

2. Értesítési logika: A Publisher végigmegy a feliratkozók listáján és meghívja az update() metódust rajtuk.

```
for (Subscriber s : subscribers) {
    s.update(this);
}
```

3. Subscriber interfész: Ez a Subscriber interface, amelyet minden megfigyelő megvalósít.

```
public interface Subscriber {
    void update(WeatherStation context); // context paraméter - 5. pont
}
```

4. Concrete Subscriber – Display: A konkrét Subscriber a Publisher-től kapott állapot alapján frissíti a megjelenítést. A Publisher objektumát (context) paraméterben kapja meg → innen kérdez le adatot.

```
public class Display implements Subscriber {

    private String name;

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

    @Override
    public void update(WeatherStation context) {  // 5. pont: context átadása
        System.out.println(
            name + " kijelző frissítve: " + context.getTemperature() + "°C"
        );
    }
}
```

5. Client: A Client hozza létre a Publisher és Subscriber példányokat, majd feliratkoztatja őket — pontosan az ábrának megfelelően.

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

        WeatherStation station = new WeatherStation();

        Subscriber d1 = new Display("Első");
        Subscriber d2 = new Display("Második");

        station.subscribe(d1);
        station.subscribe(d2);

        station.setTempe
        station.setTemperature(25.0f);
    }
}
```

A ChatGPT megkérdezése