**Tervezési minták egy OO programozási nyelvben**

Modularitás és rugalmasság biztosítása tervezési mintákkal az OOP keretében:

Orosz tamás

2024

Tartalomjegyzék

[Mi a design pattern? 2](#_Toc183966899)

[Mi alkot egy mintát? 3](#_Toc183966900)

[A minták története 3](#_Toc183966901)

[Miért érdemes megtanulni a mintákat? 3](#_Toc183966902)

[A minták kritikája 4](#_Toc183966903)

[Mankó gyenge programozási nyelvekhez 4](#_Toc183966904)

[Hatástalan megoldások 4](#_Toc183966905)

[Indokolatlan használat 4](#_Toc183966906)

[A minták osztályozása 5](#_Toc183966907)

[Alapvető minták – Idiómák 5](#_Toc183966908)

[Magas szintű minták – Architektúra minták 5](#_Toc183966909)

[Minták csoportosítása céljuk szerint 5](#_Toc183966910)

[1. Kreációs minták 5](#_Toc183966911)

[2. Strukturális minták 6](#_Toc183966912)

[3. Viselkedési minták 6](#_Toc183966913)

[A tervezési minták katalógusából néhány részletesebb leírás. 7](#_Toc183966914)

[A Singleton tervezési minta 7](#_Toc183966915)

[Builder Tervezési Minta 10](#_Toc183966916)

[Observer Tervezési Minta 13](#_Toc183966917)

[MVC Design Pattern 17](#_Toc183966918)

[Mi az MVC Design Pattern? 17](#_Toc183966919)

[Az MVC komponensei 17](#_Toc183966920)

[A komponensek közötti kommunikáció 18](#_Toc183966921)

[Példa az MVC Design Pattern használatára 18](#_Toc183966922)

# Mi a design pattern?

A tervezési minták tipikus megoldások gyakran előforduló problémákra a szoftvertervezésben. Olyan, mint egy előre elkészített tervrajz, amelyet testre szabhatsz, hogy megoldj egy ismétlődő tervezési problémát a kódban.

Nem elég azonban csak megtalálni egy mintát és bemásolni a programba, mint ahogy egy kész függvényt vagy könyvtárat használhatnál. A minta nem konkrét kódrészlet, hanem egy általános koncepció egy adott probléma megoldására. A minta részleteit követve megvalósíthatsz egy olyan megoldást, amely illeszkedik a programod sajátosságaihoz.

A mintákat gyakran összekeverik az algoritmusokkal, mivel mindkettő ismert problémák tipikus megoldásait írja le. Az algoritmus viszont mindig egyértelmű lépések sorozatát határozza meg egy cél elérése érdekében, míg a minta magasabb szintű leírást ad a megoldásról. Ugyanaz a minta eltérő kóddal valósulhat meg két különböző programban.

Egy algoritmushoz hasonlóan a minta is szemléltethető egy főzési recepttel: mindkettő konkrét lépéseket tartalmaz egy cél eléréséhez. Ugyanakkor egy minta inkább olyan, mint egy tervrajz: láthatod, hogy mi lesz az eredmény és annak jellemzői, de a megvalósítás pontos sorrendje rajtad múlik.

## Mi alkot egy mintát?

A minták leírása gyakran nagyon formális, hogy sokféle környezetben újraalkothatók legyenek. Az alábbiakban felsoroljuk, hogy milyen részeket tartalmaz általában egy minta leírása:

* **A minta célja (intent):** Röviden leírja a problémát és a megoldást.
* **Motiváció:** Részletesebben kifejti a problémát, és megmutatja, hogy a minta milyen megoldást tesz lehetővé.
* **Osztályok szerkezete:** Bemutatja a minta minden elemét és azok kapcsolatait.
* **Kódpélda:** Egy népszerű programozási nyelven készült példa megkönnyíti a minta megértését.
* **Egyéb részletek:** Egyes mintagyűjtemények tartalmazzák a minta alkalmazhatóságát, a megvalósítás lépéseit és a más mintákkal való kapcsolatait is.

## A minták története

* **Ki találta fel a mintákat?** Ez jó kérdés, de nem teljesen pontos. A tervezési minták nem rejtélyes, bonyolult fogalmak – épp ellenkezőleg. A minták tipikus megoldások az objektumorientált tervezés gyakori problémáira. Amikor egy megoldás újra és újra felbukkan különböző projektekben, valaki végül nevet ad neki és részletesen leírja. Így „fedeznek fel” egy mintát.
* A minta fogalmát először **Christopher Alexander** írta le az *A Pattern Language: Towns, Buildings, Construction* című könyvében. Ez a könyv egy „nyelvet” ír le a városi környezet tervezésére. Ennek a nyelvnek az egységei a minták. Ezek például leírhatják, milyen magasak legyenek az ablakok, hány szintes legyen egy épület, mekkorák legyenek a zöld területek egy környéken, és így tovább.
* Ezt az ötletet négy szerző, **Erich Gamma**, **John Vlissides**, **Ralph Johnson** és **Richard Helm** alkalmazta a programozás területére. 1994-ben megjelentették a *Design Patterns: Elements of Reusable Object-Oriented Software* című könyvet, amelyben a tervezési minták fogalmát programozásban használták. A könyv 23 mintát mutatott be, amelyek különböző objektumorientált tervezési problémák megoldását kínálták, és gyorsan bestsellerré vált. Hosszú címe miatt az emberek „a négyek bandájának könyveként” kezdték emlegetni, ami röviden csak „GoF könyv” lett.
* Azóta tucatnyi más objektumorientált minta is felfedezésre került. A „minták megközelítése” más programozási területeken is rendkívül népszerűvé vált, így ma már számos minta létezik az objektumorientált tervezésen kívül is.

# Miért érdemes megtanulni a mintákat?

Az igazság az, hogy akár évekig is dolgozhatsz programozóként anélkül, hogy egyetlen tervezési mintáról is hallanál. Sokan így is tesznek. Mégis, előfordulhat, hogy tudattodon kívül már használsz néhány mintát. Akkor miért érdemes külön időt szánni a megtanulásukra?

A tervezési minták olyan eszközkészletet kínálnak, amelyek bevált megoldásokat tartalmaznak a szoftvertervezés gyakori problémáira. Még ha soha nem is találkozol ezekkel a konkrét problémákkal, a minták ismerete akkor is hasznos, mert megtanít az objektumorientált tervezés alapelvei szerint gondolkodni és problémákat megoldani.

Ezen kívül a minták egy közös nyelvet is teremtenek a fejlesztőcsapatokban, ami nagyban megkönnyíti a kommunikációt. Ha például azt mondod, hogy „használjunk egy Singleton-t erre”, mindenki érteni fogja, mire gondolsz, anélkül, hogy hosszasan magyaráznod kellene. Ha ismered a minták neveit és azok jelentését, azzal gyorsabbá és hatékonyabbá teheted a munkát.

## A minták kritikája

Úgy tűnik, szinte mindenki kritizálta már a tervezési mintákat. Nézzük meg a leggyakoribb érveket a használatuk ellen!

## Mankó gyenge programozási nyelvekhez

Gyakran akkor van szükség mintákra, amikor egy programozási nyelv vagy technológia nem kínál megfelelő absztrakciós szintet. Ilyenkor a minták mankóként szolgálnak, hogy a nyelv olyan „szuperképességeket” kapjon, amelyekre amúgy nem lenne képes.

Például a *Stratégia minta* modern programozási nyelvekben egyszerűen megvalósítható egy névtelen függvénnyel (lambda).

## Hatástalan megoldások

A minták az általánosan használt megközelítéseket próbálják rendszerbe foglalni. Sokan ezt dogmaként kezelik, és a mintákat szigorúan az előírt módon alkalmazzák, anélkül hogy figyelembe vennék a projektjük konkrét igényeit.

## Indokolatlan használat

„Ha minden eszközöd egy kalapács, minden problémát szögnek látsz.”

Ez a probléma gyakran előfordul azoknál, akik épp csak megismerkedtek a mintákkal. Miután megtanulták a mintákat, hajlamosak mindenre ezeket alkalmazni, még akkor is, ha az egyszerűbb kód jobban megfelelne az adott helyzetre.

## A minták osztályozása

A tervezési minták különböznek egymástól a komplexitásuk, részletességük és abban, hogy mekkora rendszerekre alkalmazhatók. Egy jó hasonlat lehet az útépítés: egy kereszteződést biztonságosabbá tehetsz egyszerűen forgalmi lámpákkal, de akár egy többszintes csomópontot is építhetsz gyalogos aluljárókkal.

## Alapvető minták – Idiómák

A legegyszerűbb és legalacsonyabb szintű mintákat gyakran *idiómáknak* nevezik. Ezek általában csak egy adott programozási nyelvre vonatkoznak.

## Magas szintű minták – Architektúra minták

A legáltalánosabb és legmagasabb szintű minták az *architektúra minták*. Ezeket szinte bármelyik programozási nyelven meg lehet valósítani. Más mintákkal ellentétben ezek akár egy teljes alkalmazás architektúrájának megtervezésére is alkalmasak.

# Minták csoportosítása céljuk szerint

A mintákat a céljuk vagy funkciójuk alapján is kategorizálhatjuk. Három fő csoportba sorolhatók:

* **Kreációs minták:** Olyan mechanizmusokat kínálnak az objektumok létrehozására, amelyek növelik a rugalmasságot és elősegítik a meglévő kód újrahasználatát.
* **Strukturális minták:** Megmagyarázzák, hogyan lehet objektumokat és osztályokat nagyobb szerkezetekké összeállítani úgy, hogy ezek a struktúrák rugalmasak és hatékonyak maradjanak.
* **Viselkedési minták:** Az objektumok közötti hatékony kommunikációval és a felelősségek kiosztásával foglalkoznak.

**A tervezési minták katalógusa**

A tervezési minták széles körű gyűjteményt alkotnak, amelyben különböző problémák megoldására alkalmas módszereket találunk. Ezek a minták általában három fő kategóriába sorolhatók a céljuk alapján: kreációs, strukturális és viselkedési minták. Lássuk a legfontosabb mintákat röviden!

## 1. Kreációs minták

A kreációs minták az objektumok létrehozásának módjait szabályozzák, hogy elősegítsék a rugalmasságot és az újrahasználhatóságot.

* **Singleton:** Biztosítja, hogy egy osztályból csak egyetlen példány létezzen, és globálisan elérhető legyen.
* **Factory Method:** Egy interfészt biztosít az objektumok létrehozására, de az alosztályokra bízza a pontos típus meghatározását.
* **Abstract Factory:** Több, egymással összefüggő objektumcsalád létrehozását teszi lehetővé anélkül, hogy konkrét osztályokat kellene megnevezni.
* **Builder:** Összetett objektumok fokozatos létrehozását támogatja egy rugalmas lépéssorozat segítségével.
* **Prototype:** Lehetővé teszi objektumok másolását anélkül, hogy azok konkrét osztályaira hivatkoznánk.

## 2. Strukturális minták

A strukturális minták az objektumok és osztályok összeszervezésével foglalkoznak, hogy hatékony és könnyen bővíthető struktúrákat hozzanak létre.

* **Adapter:** Egy meglévő osztály interfészét átalakítja úgy, hogy az kompatibilis legyen egy másik interfésszel.
* **Bridge:** Különválasztja az absztrakciót a megvalósítástól, lehetővé téve, hogy egymástól függetlenül változhassanak.
* **Composite:** Lehetővé teszi, hogy objektumokat hierarchikus szerkezetben kezeljünk, mint például fákat.
* **Decorator:** Dinamikusan bővíti egy objektum funkcionalitását anélkül, hogy módosítaná annak kódját.
* **Facade:** Egy egyszerűsített interfészt biztosít egy komplex rendszerhez.
* **Flyweight:** Több hasonló objektum közös erőforrások megosztásával való hatékony kezelésére szolgál.
* **Proxy:** Egy másik objektum helyettesítéseként működik, például hozzáférés-szabályozáshoz vagy teljesítmény-optimalizáláshoz.

## 3. Viselkedési minták

A viselkedési minták az objektumok közötti interakcióval és a felelősségek elosztásával foglalkoznak.

* **Chain of Responsibility:** Egy kérést több objektum láncban kezel, amíg valaki meg nem oldja.
* **Command:** Egy kérést objektummá alakít, lehetővé téve a kérések paraméterezését, naplózását és visszavonását.
* **Interpreter:** Egy adott nyelvtan szabályait valósítja meg, lehetővé téve az adott nyelv parancsainak értelmezését.
* **Iterator:** Sorozatok elemeinek bejárására szolgáló szabványos módszert biztosít.
* **Mediator:** Egy közvetítő objektum egyszerűsíti az objektumok közötti kommunikációt.
* **Memento:** Lehetővé teszi az objektumok állapotának mentését és visszaállítását anélkül, hogy azok belső részleteihez hozzáférnénk.
* **Observer:** Az egyik objektum állapotának változásait több másik objektum figyeli és reagálja le.
* **State:** Egy objektum viselkedése a belső állapotától függ, és dinamikusan változhat.
* **Strategy:** Több algoritmus közötti választást tesz lehetővé egy adott probléma megoldására.
* **Template Method:** Az algoritmus vázát egy absztrakt osztályban definiálja, de egyes lépéseit az alosztályokra bízza.
* **Visitor:** Lehetővé teszi, hogy egy műveletet futtassunk különböző típusú objektumokon, anélkül hogy azok osztályait módosítanánk.

Ez a katalógus jól mutatja, hogy a tervezési minták különböző szinteken és helyzetekben nyújtanak hatékony megoldásokat. Az ismeretük jelentősen megkönnyítheti a fejlesztést és az együttműködést a csapatban!

# A tervezési minták katalógusából néhány részletesebb leírás.

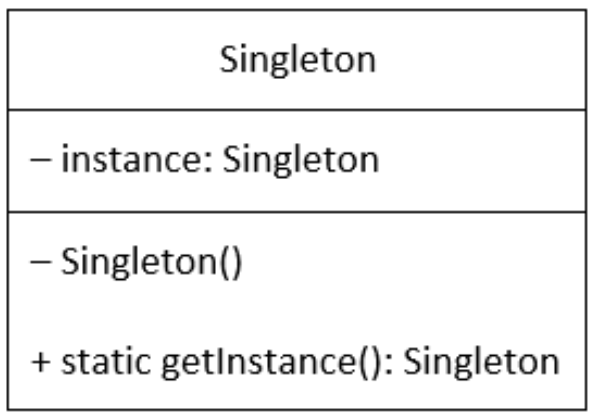
## A Singleton tervezési minta

A **Singleton** tervezési minta egy kreációs minta, amelynek célja, hogy egy osztályból csak egyetlen példány jöjjön létre, és ehhez az objektumhoz globális hozzáférési pontot biztosítson.

Egy tipikus példa erre a Java-ban a **Calendar** osztály, ahol nem lehet közvetlenül példányt létrehozni. Ehelyett az osztály a saját **getInstance()** metódusát biztosítja az objektum elérésére.

**Egy Singleton mintát használó osztály jellemzői:**

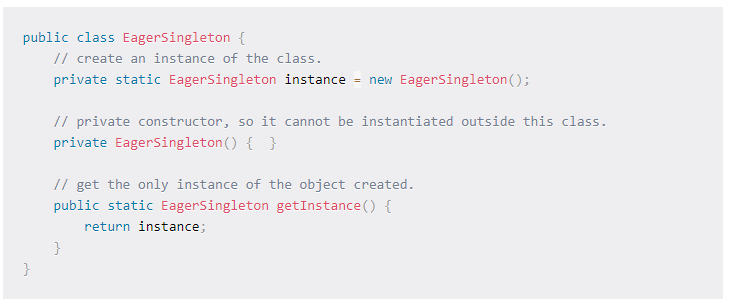
1. **Privát konstruktor**  
   Az osztály konstruktora privát, hogy megakadályozza az objektum példányosítását külső kódból.
2. **Statikus osztályváltozó az objektum tárolására**  
   Egy statikus változó tárolja az egyetlen példányt, amelyet az osztály létrehoz.
3. **Statikus hozzáférési metódus (getInstance)**  
   Egy nyilvános statikus metódus biztosítja az objektumhoz való hozzáférést. Ha az objektum még nem létezik, akkor itt jön létre; ha már létezik, akkor ugyanazt a példányt adja vissza.



**Singleton Minta Különböző Megvalósításai**

A Singleton mintát többféleképpen lehet megvalósítani, attól függően, hogy mikor és hogyan jön létre az egyetlen példány, illetve hogyan kezeljük a párhuzamos szálakat. Lássuk a három leggyakoribb megközelítést:

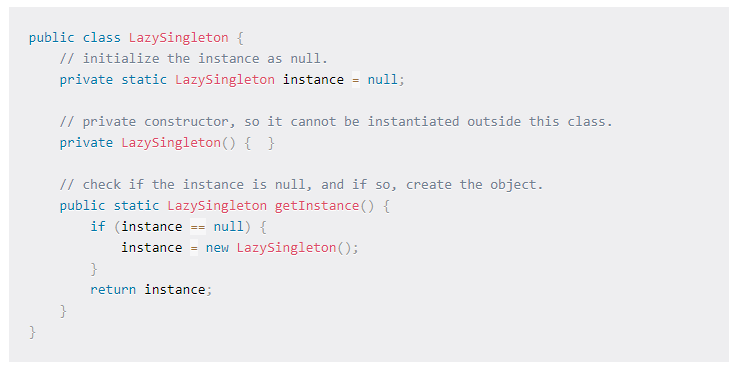
**Eager Beaver**



Ez a fajta példányosítás az osztály betöltésekor történik meg, mivel a példány létrehozása az osztályon kívül, nem pedig egy metódusban zajlik. Ennek komoly hátránya, hogy az objektum akkor is létrejön, ha az alkalmazásban végül egyáltalán nem használják az adott osztályt. Az ilyen esetek kezelésére alternatív megoldásként a késleltetett példányosítást (Lazy Instantiation) szokták alkalmazni.

**Lazy Days**

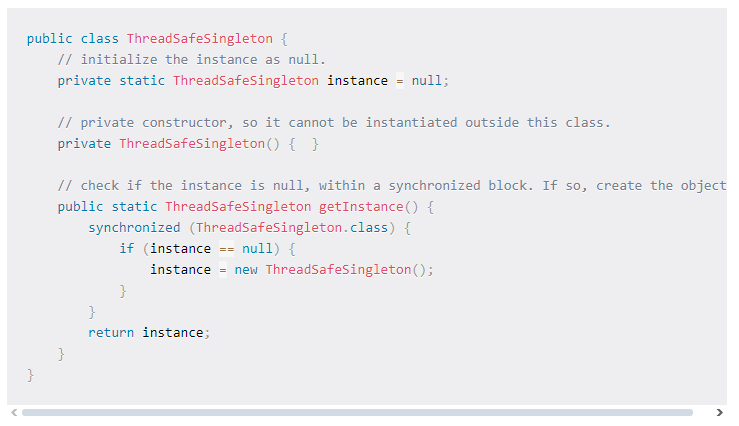
Nincs nagy különbség a fenti megvalósításhoz képest. A fő eltérés, hogy a statikus változó kezdetben **null** értékre van beállítva, és csak akkor történik meg a példányosítás a **getInstance()** metódusban, ha – és csak akkor – az **instance** változó továbbra is **null** a vizsgálat idején.



Ez megold egy problémát, de egy másik még mindig fennáll. Mi történik, ha két különböző kliens egyszerre, pontosan ugyanabban a millimásodpercben fér hozzá a Singleton osztályhoz? Ebben az esetben mindkét kliens ugyanakkor ellenőrzi, hogy az **instance** változó null-e, és mindketten igaznak találják, így mindkét kéréshez külön példányokat hoznak létre. Ennek a problémának a megoldására a szálbiztos példányosítást (Thread Safe instantiation) kell alkalmazni.

**(Thread) Safety is Key**

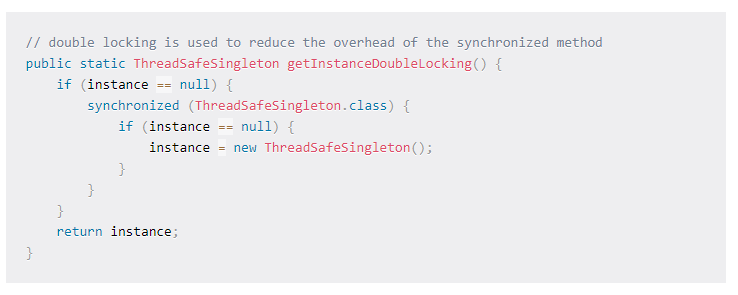
A Java-ban a **synchronized** kulcsszót metódusokon vagy objektumokon használják a szálbiztosság megvalósítására, biztosítva, hogy egyszerre csak egy szál férhessen hozzá egy adott erőforráshoz. Az osztály példányosítása egy **synchronized** blokkba kerül, így a metódust egy adott időpontban csak egy kliens érheti el, megakadályozva, hogy párhuzamosan több szál is példányosítson egy objektumot.



A **synchronized** metódus használata magas költséggel jár, ami csökkenti az egész művelet teljesítményét.

Például, ha az **instance** változó már példányosítva van, akkor minden alkalommal, amikor egy kliens hozzáfér a **getInstance()** metódushoz, a szinkronizált metódus lefut, ami rontja a teljesítményt. Ez akkor történik, amikor ellenőrizzük, hogy az **instance** változó értéke **null**-e. Ha igen, akkor elhagyja a metódust.

Ennek a költségnek a csökkentése érdekében a **kettős zár** (double locking) technikát alkalmazzák. Az ellenőrzést már a szinkronizált metódus előtt is végrehajtják, és csak akkor fut le a szinkronizált metódus, ha az érték valóban **null**.



## Builder Tervezési Minta

A **Builder** tervezési minta egy strukturális minta, amelyet akkor használunk, amikor egy összetett objektumot kell létrehozni, de nem szeretnénk, hogy az objektum konstruktora túl bonyolult legyen, és nem akarjuk, hogy az összes paraméter egyszerre legyen átadva. A Builder minta lehetővé teszi, hogy az objektumot lépésről lépésre építsük fel, miközben elrejtjük az összetett konstrukciós logikát.

**Miért használjuk?**

* **Tisztább kód:** Ha egy objektum létrehozása sok paramétert igényel, a kód nehezen olvasható és karbantartható lehet. A Builder minta segít elválasztani az objektum felépítését az objektum használatától.
* **Immutabilitás:** A Builder minta lehetővé teszi, hogy az objektumokat módosíthatatlanul hozzuk létre.
* **Flexibilitás:** Lehetővé teszi a különböző konfigurációk létrehozását anélkül, hogy túl sok konstruktorváltozatot kellene definiálni.

**Builder Minta Alapfelépítése**

A Builder minta gyakran három fő komponensből áll:

1. **Product (Termék):** Az az osztály, amelyet a Builder minta segítségével építünk.
2. **Builder:** Az osztály, amely a terméket építi fel. A Builder az összes szükséges paramétert átveszi, majd létrehozza a terméket.
3. **Director:** Az osztály, amely a Builder-t irányítja, és meghatározza a lépéseket az objektum felépítéséhez.

**Példa:**

Vegyünk egy példát egy **Autó** osztályról, amelyet különböző konfigurációkkal szeretnénk létrehozni.

**1. Autó osztály (Product)**

public *class* Car {

*private* String engine;

*private* String wheels;

*private* String color;

*private* int doors;

*// Konstruktorról nem beszélünk most, mivel a Builder fogja építeni.*

*// Getters*

*public* String getEngine() {

*return* engine;

    }

*public* String getWheels() {

*return* wheels;

    }

*public* String getColor() {

*return* color;

    }

*public* int getDoors() {

*return* doors;

    }

*// Builder class*

*public* *static* *class* Builder {

*private* String engine;

*private* String wheels;

*private* String color;

*private* int doors;

*// Konstruktorok a különböző paraméterekhez*

*public* Builder setEngine(String *engine*) {

            this.engine = *engine*;

*return* this;

        }

*public* Builder setWheels(String *wheels*) {

            this.wheels = *wheels*;

*return* this;

        }

*public* Builder setColor(String *color*) {

            this.color = *color*;

*return* this;

        }

*public* Builder setDoors(int *doors*) {

            this.doors = *doors*;

*return* this;

        }

*// A végső lépés, ami létrehozza az autót*

*public* Car build() {

*return* *new* Car(this);

        }

    }

*// Privát konstruktor, amit csak a Builder hívhat*

*private* Car(Builder *builder*) {

        this.engine = *builder*.engine;

        this.wheels = *builder*.wheels;

        this.color = *builder*.color;

        this.doors = *builder*.doors;

    }

}

**2. Például egy autó létrehozása Builder segítségével:**

public *class* Main {

*public* *static* void main(String[] *args*) {

*// Builder segítségével egy autó létrehozása*

        Car car = *new* Car.Builder()

                .setEngine("V8")

                .setWheels("Alloy")

                .setColor("Red")

                .setDoors(4)

                .build();

*// Az autó adatai*

        System.out.println("Car Engine: " + car.getEngine());

        System.out.println("Car Wheels: " + car.getWheels());

        System.out.println("Car Color: " + car.getColor());

        System.out.println("Car Doors: " + car.getDoors());

    }

}

**Mi történik itt?**

* Az **Autó** osztály tartalmaz egy **Builder** osztályt, amely minden paramétert lépésről lépésre állít be.
* A **setEngine**, **setWheels**, **setColor** és **setDoors** metódusok láncolhatók, mivel mindegyik visszaadja a **Builder** példányát. Ez lehetővé teszi, hogy a kód olvasható és tiszta maradjon.
* A végén a **build()** metódus hívása építi meg az autót az összes paraméterrel.

**Előnyök**

1. **Olvashatóság:** A kód világos, mivel minden paraméter külön metódusokkal van beállítva.
2. **Flexibilitás:** A Builder lehetővé teszi, hogy az objektumokat többféle módon is létrehozhassuk anélkül, hogy túl sok konstruktorváltozatot kellene definiálni.
3. **Immutabilitás:** Az objektumok létrehozása után nem változtathatók meg, mivel nincs setter metódusuk.

**Hátrányok**

1. **Túlzottan komplex objektumok esetén nehézségek lehetnek:** Ha túl sok paraméter van, a Builder osztály is túl nagyra nőhet.
2. **Túl sok kód:** Az egyszerű objektumok esetén a Builder minta feleslegessé válhat, és többletkódot eredményezhet.

## Observer Tervezési Minta

Az **Observer** tervezési minta egy viselkedési minta, amely a "kiértesítési" vagy "figyelés" mechanizmusokra épít. Az Observer minta lehetővé teszi, hogy egy objektum (az úgynevezett **Subject**) értesítse a hozzá tartozó **Observer** objektumokat (vagy figyelőket), ha valamilyen állapotváltozása történik. Ezzel biztosítható, hogy több objektum is reagálni tudjon egy másik objektum állapotváltozására anélkül, hogy szoros kapcsolat lenne közöttük.

**Miért használjuk az Observer mintát?**

* **Decoupling (Leválasztás):** A minta lehetővé teszi, hogy az objektumok ne legyenek szorosan összekapcsolva. Az Observer osztályok nem ismerik közvetlenül a Subject osztályt, csak annak egy publikus metódusát, amelyet akkor hívnak meg, amikor az állapot változik.
* **Egyszerű kiértesítés:** Az Observer minta egyszerűvé teszi a több objektum közötti értesítések kezelését, amikor egy objektum állapota megváltozik.
* **Reaktivitás:** Ez a minta segíthet reaktív rendszerek felépítésében, ahol különböző komponensek reagálnak más komponensek állapotváltozásaira.

**A minta elemei:**

1. **Subject (Téma):** Az az objektum, amelynek az állapotában változás történik. Ez az objektum értesíti a figyelőket (Observers).
2. **Observer (Figyelő):** Azok az objektumok, amelyek érdeklődnek a Subject állapotának változásai iránt. Az Observerek regisztrálják magukat a Subject objektumba, és értesítést kapnak, amikor annak állapota megváltozik.
3. **ConcreteSubject (Konkrét téma):** Az a konkrét Subject osztály, amelyet az Observerek figyelnek.
4. **ConcreteObserver (Konkrét figyelő):** Az Observer konkrét implementációja, amely reagál a változásokra.

**Példa:**

Vegyünk egy egyszerű példát, ahol egy **Újságkiadó** (Subject) értesíti a **Felhasználókat** (Observers) a legfrissebb hírekről.

**1. Observer és Subject interfészek**

Először definiáljuk az Observer és Subject interfészeket, amelyeket a konkrét osztályok implementálni fognak:

*// Observer interfész*

*public* *interface* Observer {

    void update(String *news*);

}

*// Subject interfész*

*public* *interface* Subject {

    void registerObserver(Observer *observer*);

    void removeObserver(Observer *observer*);

    void notifyObservers();

}

**2. Konkrét Subject (Újságkiadó)**

Ez a **ConcreteSubject** osztály valósítja meg a Subject interfészt, és tárolja a figyelők listáját. A változások történtekor az update metódusokat hívja.

import java.util.ArrayList;

import java.util.List;

*// ConcreteSubject osztály*

*public* *class* NewsAgency *implements* Subject {

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

*private* String latestNews;

*// Hozzáad egy új figyelőt*

    @Override

*public* void registerObserver(Observer *observer*) {

        observers.add(*observer*);

    }

*// Eltávolít egy figyelőt*

    @Override

*public* void removeObserver(Observer *observer*) {

        observers.remove(*observer*);

    }

*// Értesíti az összes regisztrált figyelőt*

    @Override

*public* void notifyObservers() {

*for* (Observer observer *:* observers) {

            observer.update(latestNews);

        }

    }

*// Beállítja az új hírt, és értesíti a figyelőket*

*public* void setLatestNews(String *news*) {

        this.latestNews = *news*;

        notifyObservers();

    }

}

**3. Konkrét Observer (Felhasználó)**

Ez a **ConcreteObserver** osztály az Observer interfészt valósítja meg, és reagál a hírek változására.

*// ConcreteObserver osztály*

*public* *class* User *implements* Observer {

*private* String name;

*public* User(String *name*) {

        this.name = *name*;

    }

*// Reagál a hír frissítésére*

    @Override

*public* void update(String *news*) {

        System.out.println(name + " received news: " + *news*);

    }

}

**4. Példa használat:**

A következő kódban bemutatjuk, hogyan használhatjuk a **NewsAgency** (újságkiadó) osztályt és hogyan regisztrálhatunk **Felhasználókat** (figyelőket), akik értesítést kapnak a legfrissebb hírekről.

public *class* Main {

*public* *static* void main(String[] *args*) {

*// Létrehozzuk az újságkiadót (Subject)*

        NewsAgency newsAgency = *new* NewsAgency();

*// Létrehozzuk a felhasználókat (Observers)*

        User user1 = *new* User("Alice");

        User user2 = *new* User("Bob");

*// Regisztráljuk a felhasználókat*

        newsAgency.registerObserver(user1);

        newsAgency.registerObserver(user2);

*// Új hír beállítása, amely értesíti a felhasználókat*

        newsAgency.setLatestNews("Breaking News: Java Design Patterns are awesome!");

*// Egy felhasználó leiratkozása a hírek fogadásáról*

        newsAgency.removeObserver(user2);

*// Újabb hír, amit már csak Alice fog megkapni*

        newsAgency.setLatestNews("Breaking News: Observer Pattern is easy to understand!");

    }

}

**Mi történik a kódban?**

1. Létrehozzuk a **NewsAgency** (újságkiadó) osztályt, amely a **Subject**.
2. Létrehozzuk két **User** (felhasználó) objektumot, akik az **Observer** osztályokat reprezentálják.
3. Regisztráljuk őket az **újságkiadó** figyelőiként a **registerObserver** metódus segítségével.
4. Beállítunk egy új hírt a **setLatestNews** metódusban, amely értesíti a regisztrált figyelőket.
5. Az egyik felhasználó leiratkozik a hírek fogadásáról a **removeObserver** metódus segítségével.
6. Egy új hír frissítésével csak a maradék figyelők kapják meg az értesítést.

**Kimenet:**



**Előnyök**

* **Decoupling:** A Subject és az Observerek nem függnek közvetlenül egymástól. Az Observerek nem tudják, hogy mi történik a Subject-ben, csak azt tudják, hogy értesítést kell kapniuk.
* **Egyszerűsített értesítések:** Egy objektum állapotának megváltozása egyszerűen értesítheti a hozzá tartozó Observereket.
* **Skálázhatóság:** Könnyen bővíthető, ha több Observert kell hozzáadni, anélkül hogy módosítani kellene a Subject osztályt.

**Hátrányok**

* **Túl sok figyelő esetén teljesítménybeli problémák adódhatnak**, mivel minden egyes állapotváltozás értesíti az összes regisztrált figyelőt.
* **Felesleges értesítések:** Ha nem megfelelően van kezelve, akkor az Observerek túl gyakran kaphatnak értesítéseket, még akkor is, ha azokat nem használják.

# MVC Design Pattern

Az **MVC (Model-View-Controller)** tervezési minta egy olyan szoftver architektúra minta, amely három fő komponensre osztja az alkalmazást: **Model**, **View**, és **Controller**. Ez a minta segít az alkalmazás kódjának kezelésében és karbantartásában, mivel a három komponens különválasztása biztosítja, hogy könnyebben módosíthassuk az alkalmazás egy-egy részét anélkül, hogy a többi részre hatással lennénk. Emellett elősegíti a komponensek újrahasználhatóságát, és modulárisabb megközelítést biztosít a szoftverfejlesztésben.

## Mi az MVC Design Pattern?

A **Model View Controller** (MVC) tervezési minta egy olyan szoftver architektúra, amely az alkalmazást három fő részre bontja: **Model** (adatok), **View** (megjelenítés) és **Controller** (vezérlő). E három komponens közötti szétválasztásnak köszönhetően az alkalmazás karbantartása és bővítése könnyebbé válik, mivel az egyes komponensek módosítása nem érinti a többi komponenst.

Ez a minta az alkalmazás logikáját, az adatokat és a felhasználói interakciót külön kezeli, így az egyes részek könnyebben kezelhetők és módosíthatók.

## Az MVC komponensei

**1. Model**

A **Model** felelős az alkalmazás adatainak és üzleti logikájának kezeléséért. A modellek az alkalmazás állapotát képviselik, és mindent, ami az adatokkal kapcsolatos, ide helyezünk. A Model nem tud semmit a View-ról vagy a Controller-ről, csupán adatokat biztosít a Controller számára, aki azt továbbítja a View-nak.

**Példa**: Egy User osztály, amely kezeli a felhasználói adatokat, például név és email cím, valamint biztosítja az adatok lekérdezését és frissítését.

**2. View**

A **View** felelős az adatok vizuális megjelenítéséért a felhasználó számára. A View kizárólag az alkalmazás felhasználói felületét (UI) jeleníti meg, de nem tartalmaz üzleti logikát. Az adatokat a View a Model-től kapja, és azokat egy érthető formában jeleníti meg a felhasználónak.

**Példa**: Egy HTML oldal, amely a felhasználói adatokat jeleníti meg egy táblázatban, vagy egy grafikus felhasználói felület (GUI), amely a Model által szolgáltatott adatokat vizualizálja.

**3. Controller**

A **Controller** közvetítő szerepet tölt be a Model és a View között. A Controller kezeli a felhasználói inputokat, és elvégzi azokat a műveleteket, amelyek az adatok frissítéséhez vagy módosításához szükségesek. Miután a Controller frissítette az adatokat a Model-ben, értesíti a View-t, hogy frissítse a megjelenítést.

**Példa**: Egy UserController osztály, amely a felhasználó által megadott adatokat fogadja, ellenőrzi, és frissíti a Model-t, majd a View-t frissíti az új adatokkal.

## A komponensek közötti kommunikáció

A következő lépéseken keresztül történik az MVC komponensek közötti kommunikáció:

1. **Felhasználói interakció a View-val**: A felhasználó interakcióba lép a View-val (pl. egy gomb megnyomása, vagy szöveg bevitele egy űrlapba).
2. **View fogadja a felhasználói inputot**: A View továbbítja az inputot a Controller felé.
3. **Controller feldolgozza a felhasználói inputot**: A Controller a View-tól kapott input alapján végrehajtja a szükséges műveleteket, például frissíti a Model-t.
4. **Controller frissíti a Model-t**: A Controller frissíti az adatokat a Model-ben a felhasználói input vagy más logika alapján.
5. **Model értesíti a View-t a változásról**: Ha a Model változik, értesíti a View-t, hogy az frissíthesse a megjelenítést.
6. **View kér adatokat a Model-től**: A View a szükséges adatokat a Model-től kéri le.
7. **Controller frissíti a View-t**: A Controller az új adatok alapján frissíti a View-t.
8. **View frissíti a felhasználói felületet**: A View megjeleníti a frissített UI-t a felhasználó számára.

## Példa az MVC Design Pattern használatára

Nézzünk egy egyszerű példát a fenti tervezési minta alkalmazására, egy diák adatait kezelő alkalmazást.

#### Model (Student osztály)

A **Model** a diák adatait tartalmazza, mint a neve és a nyilvántartási száma.

class Student **{**

private String rollNo**;**

private String name**;**

public String getRollNo**()** **{**

**return** rollNo**;**

**}**

public void setRollNo**(**String rollNo**)** **{**

**this.**rollNo **=** rollNo**;**

**}**

public String getName**()** **{**

**return** name**;**

**}**

public void setName**(**String name**)** **{**

**this.**name **=** name**;**

**}**

**}**

**View (StudentView osztály)**

A **View** felelős a diák adatait megjeleníteni.

class StudentView **{**

public void printStudentDetails**(**String studentName**,** String studentRollNo**)** **{**

System**.**out**.**println**(**"Student:"**);**

System**.**out**.**println**(**"Name: " **+** studentName**);**

System**.**out**.**println**(**"Roll No: " **+** studentRollNo**);**

**}**

**}**

**Controller (StudentController osztály)**

A **Controller** felelős az adatok frissítéséért és a nézetek frissítéséért.

class StudentController **{**

private Student model**;**

private StudentView view**;**

public StudentController**(**Student model**,** StudentView view**)** **{**

**this.**model **=** model**;**

**this.**view **=** view**;**

**}**

public void setStudentName**(**String name**)** **{**

model**.**setName**(**name**);**

**}**

public String getStudentName**()** **{**

**return** model**.**getName**();**

**}**

public void setStudentRollNo**(**String rollNo**)** **{**

model**.**setRollNo**(**rollNo**);**

**}**

public String getStudentRollNo**()** **{**

**return** model**.**getRollNo**();**

**}**

public void updateView**()** **{**

view**.**printStudentDetails**(**model**.**getName**(),** model**.**getRollNo**());**

**}**

**}**

**Fő program (MVCPattern osztály)**

Az alkalmazás futtatásához az alábbi kódot használjuk.

public class MVCPattern **{**

public static void main**(**String**[]** args**)** **{**

Student model **=** retriveStudentFromDatabase**();**

StudentView view **=** **new** StudentView**();**

StudentController controller **=** **new** StudentController**(**model**,** view**);**

controller**.**updateView**();**

controller**.**setStudentName**(**"Vikram Sharma"**);**

controller**.**updateView**();**

**}**

private static Student retriveStudentFromDatabase**()** **{**

Student student **=** **new** Student**();**

student**.**setName**(**"Lokesh Sharma"**);**

student**.**setRollNo**(**"15UCS157"**);**

**return** student**;**

**}**

**}**

Kimenet:

Student**:**

Name**:** Lokesh Sharma

Roll No**:** 15UCS157

Student**:**

Name**:** Vikram Sharma

Roll No**:** 15UCS157

**Mikor használd az MVC Design Pattern-t?**

* **Bonyolult alkalmazások**: Ha az alkalmazás sok funkcióval és felhasználói interakcióval rendelkezik (például e-kereskedelmi oldalak), az MVC segít a kód rendszerezésében és a komplexitás kezelésében.
* **Gyakori UI változások**: Ha a felhasználói felület gyakori frissítést igényel, az MVC lehetővé teszi a View módosítását anélkül, hogy az alapvető logikát befolyásolná.
* **Komponens újrahasználhatóság**: Ha más projektekben is szeretnéd újrahasználni az alkalmazásod egyes részeit, az MVC moduláris felépítése megkönnyíti ezt.
* **Tesztelési igények**: Az MVC támogatja az alapos tesztelést, mivel minden komponens külön tesztelhető.

**Mikor ne használd az MVC Design Pattern-t?**

* **Egyszerű alkalmazások**: Ha az alkalmazás egyszerű és nem rendelkezik sok funkcióval, az MVC túlzott komplexitást adhat hozzá.
* **Valós idejű alkalmazások**: Az MVC nem ideális azonnali frissítéseket igénylő alkalmazásokhoz, például online játékokhoz vagy chat alkalmazásokhoz.
* **Szorosan összekapcsolt UI és logika**: Ha a UI és az üzleti logika szoros kapcsolatban van, az MVC túlságosan bonyolulttá válhat.
* **Korlátozott erőforrások**: Kisebb csapatok vagy kevésbé tapasztalt fejlesztők esetében egyszerűbb megoldások gyorsabb fejlesztést eredményezhetnek.