Tervezési minták egy OO programozási nyelvben.

Az MVC (Model-View-Controller) Architektúra
Az **MVC** egy széles körben elterjedt tervezési minta, amely segít a szoftverek felépítésében és karbantartásában. Az MVC három fő komponensből áll: **Model**, **View**, és **Controller**. Ezek a részek különböző szerepeket töltenek be, és lehetővé teszik a kód tiszta elválasztását, ami elősegíti az alkalmazás egyszerűbb fejlesztését és módosítását.

Az MVC Részei:

Model (Modell)
A Model az alkalmazás adatainak logikáját képviseli. Ez felel az adatok tárolásáért, kezeléséért, valamint a logikai műveletek végrehajtásáért. Például egy JavaFX alkalmazásban a Model lehet egy Course vagy Student osztály, amely objektumok tulajdonságait és viselkedését definiálja.
Fő funkciói:

- Adatbázis-műveletek végrehajtása.
- Adatvalidáció.
- Üzleti logika megvalósítása.

View (Nézet)
A View felel az adatok megjelenítéséért a felhasználó számára. Ez a rész határozza meg az alkalmazás felhasználói felületét (UI). A View általában nem tartalmaz logikát, csak az adatokat jeleníti meg.

Controller (Vezérlő)
A **Controller** kezeli a felhasználói interakciókat, feldolgozza a bemeneteket, és frissíti a Modelt vagy a View-t. Ez köti össze a View-t a Modellel, és biztosítja, hogy a felhasználói műveletek megfelelően hajtsák végre a kívánt logikát.
Fő szerepe:

- Felhasználói események kezelése (pl. gombnyomások).
- Adatok frissítése a Modelben.
- A View frissítésének kezdeményezése.

Az MVC tehát ideális választás a nagyobb, komplex alkalmazások fejlesztéséhez, mivel biztosítja a kód strukturált felépítését és hosszú távú karbantarthatóságát.

A Singleton Tervezési Minta
A **Singleton** egy gyakran használt tervezési minta, amely biztosítja, hogy egy osztályból csak egyetlen példány jöjjön létre az alkalmazás futása során. Ez a minta különösen hasznos olyan helyzetekben, amikor központi hozzáférésre van szükség egyetlen erőforráshoz vagy konfigurációs adathoz.

Singleton Alapelvei:

- Egyetlen Példány: Az osztály biztosítja, hogy csak egy példánya létezzen.
- Globális Hozzáférés: Az osztálypéldányhoz globálisan, bárhonnan hozzá lehet férni.
- Példány Létrehozása: Az osztály saját maga hozza létre a példányt, ha az még nem létezik.

Singleton Típusai:
Lusta Inicializáció (Lazy Initialization): A példány csak akkor jön létre, amikor először szükség van rá.
Korai Inicializáció (Eager Initialization): Az osztály betöltésekor azonnal létrejön a példány.
Thread-Safe Singleton: Többszálú környezetben biztosítja, hogy csak egy példány jöjjön létre.

Előnyök:

- **Egyszerű hozzáférés:** Az alkalmazás bármely pontjáról elérhető az osztály példánya.
- **Erőforrás-megtakarítás:** Biztosítja, hogy csak egy példány használja az erőforrást.

Hátrányok:

- **Globális állapot:** Nehezebb lehet tesztelni, mivel a globális állapot bonyolultabbá teheti a kódot.
- **Többszálúság:** Gondoskodni kell a szálbiztonságról, ha az alkalmazás több szálon fut.

A Singleton minta tehát hasznos eszköz a szoftverfejlesztésben, különösen olyan helyzetekben, amikor biztosítani kell, hogy egy osztályból csak egy példány legyen az alkalmazás életciklusa során.

A Felelősségi Lánc (Chain of Responsibility) Tervezési Minta
A **Chain of Responsibility** egy viselkedési tervezési minta, amely lehetővé teszi, hogy egy kérést egy sor osztály vagy objektum kezeljen, anélkül, hogy a kérőt a pontos kezelőhöz kellene kötni. A kérést láncban továbbítják a kezelők között, amíg az egyik kezelő el nem végzi a szükséges műveletet.

Fő Elv:
A kérés feldolgozása egy lánc mentén történik. Minden egyes láncszem (kezelő) eldöntheti, hogy:

- **Feldolgozza** a kérést.
- **Továbbítja** azt a következő láncszemhez.

Ez a megközelítés lazán kapcsolt rendszereket eredményez, amelyek könnyen bővíthetők és módosíthatók.

Fontos Elemei:

- **Absztrakt kezelő (Handler):** Meghatározza a közös interfészt és a következő kezelő referenciáját.
- **Konkrét kezelők (Concrete Handler):** Implementálják a kezelési logikát, és továbbadják a kérést, ha nem tudják kezelni.
- **Kliens:** Létrehozza a kezelők láncát, és elküldi a kérést a lánc első elemének.

Előnyök:

- **Laza csatolás:** A kérő fél nem tudja, melyik kezelő fogja feldolgozni a kérést.
- **Könnyen bővíthető:** Új kezelők hozzáadása egyszerű, anélkül, hogy a meglévő kódot módosítani kellene.
- **Rugalmasság:** A lánc dinamikusan összeállítható, a futásidejű konfigurációtól függően.

Hátrányok:

- **Bonyolultság:** A lánc túl hosszú lehet, és nehéz követni, hogy melyik kezelő felel a kérés kezeléséért.
- **Megállási probléma:** Ha egyik kezelő sem kezeli a kérést, akkor az a lánc végén feldolgozatlan maradhat.

A Chain of Responsibility tehát ideális megoldás olyan helyzetekben, amikor egy kérés kezelését rugalmasan szeretnénk megosztani különböző komponensek között, anélkül, hogy a kérőt szorosan hozzá kellene kötni a kezelőkhöz.

A Kompozit (Composite) Tervezési Minta
A **Composite** egy szerkezeti tervezési minta, amely lehetővé teszi, hogy az objektumokat **fa** szerkezetbe szervezzük. Ez a minta olyan helyzetekben hasznos, amikor az **egyedi objektumokat** és azok **összetett csoportjait** ugyanúgy szeretnénk kezelni. A kompozit minta leegyszerűsíti a hierarchikus struktúrák kezelését, mivel lehetővé teszi, hogy a kliens ugyanazzal az interfésszel dolgozzon, függetlenül attól, hogy egy önálló objektumról vagy egy összetett szerkezetről van szó.

Fő Koncepció:

- **Egyedi komponens (Leaf):** Az egyszerű objektumokat képviseli, amelyeknek nincsenek további gyermekei.
- **Összetett komponens (Composite):** Olyan objektum, amely más komponenseket (leveleket vagy további összetett komponenseket) tartalmaz.
- **Egységes interfész:** Az egyedi és összetett objektumok ugyanazt az interfészt implementálják, így a kliens kódnak nem kell különbséget tenni közöttük.

Fontos Elemei:

1. **Komponens (Component):** Egy absztrakt osztály vagy interfész, amely definiálja a fa struktúrában lévő objektumok közös műveleteit.
2. **Levél (Leaf):** Egyedi objektumokat reprezentál, amelyek nem tartalmaznak további komponenseket.
3. **Kompozit (Composite):** Összetett objektumokat képvisel, amelyek gyermek komponenseket tárolnak, és ezeken műveleteket hajtanak végre.

Előnyök:

- **Egyszerűsített kód:** A kliens kód nem tesz különbséget az egyedi és az összetett objektumok között.
- **Fa szerkezet:** Természetesen kezeli a hierarchikus struktúrákat, például fájlrendszereket vagy grafikus felületeket.
- **Rugalmasság:** Könnyen bővíthető új típusú komponensekkel.

Hátrányok:

- **Komplexitás:** A rendszer bonyolultabbá válhat, ha túl sok komponens van.
- **Típusbiztonság:** A különböző típusú objektumok közös interfésze miatt az egyedi funkciók elérhetősége korlátozott lehet.

A Composite minta tehát ideális megoldás, amikor összetett hierarchikus struktúrákat szeretnénk kezelni, és azt akarjuk, hogy a kliens számára átlátható legyen az egyedi és az összetett objektumok kezelése.