# Tervezési minták

Készitette: Vaskó Dániel

A tervezési minták angolul „design pattern” olyan megoldási módszer, amely ismétlődő problémákra ad megoldást.

A szoftverfejlesztés során egyre bonyolultabb rendszerek megalkotása válik szükségessé. Az ilyen komplexitás kezeléséri egy eszköz és az egyre gyakrabban előforduló problémák megoldását segíti általános irányelvek segítségével emiatt hatékony és strukturált a program fejlesztése. Elősegíti, hogy a kód tisztább és átláthatóbb valamint az újrahasználhatóság szempontjából is kedvező legyen. A minták ismeretével megérthetőbb a rendszer ezért ezek alkalmazása kulcsfontosságú lehet egy sikeres szoftverfejlesztési folyamatban.

A tervezési minták három fő típusra oszthatók:

1. **creational** (gyártási)
2. **structural** (szerkezeti)
3. **behavioral** (viselkedési)

## Gyártási minta:

A gyártási minták a példányosítási folyamatot úgy segítik elő, hogy az elkülönítik a példányosítás módját magától a példányosítási folyamat körülményeitől. Az, hogy pontosan mi jön létre, hogyan történik a létrehozás, vagy hogy ki végzi el, nem lényeges – csupán az a fontos, hogy mi a végül létrejövő objektum.

A **Factory Method** egy széles körben alkalmazott minta, az **Abstract Factory** rugalmas megoldást kínál, ugyanakkor egy bonyolultabb osztályhierarchia létrehozásával éri el célját, míg a **Singleton** minta lehetővé teszi, hogy egy osztályból csupán egyetlen példány jöjjön létre, amelyet mindenki elérhet, aki szükségét érzi, globálisan.

A Factory Method lényege, hogy objektumot hozunk létre úgy, hogy a kliens ne legyen terhelve a létrehozás részleteivel. Az ősosztály meghatározza az összes általános viselkedést, de a példányosítás részleteit az alosztályoknak adja át. Erre a mintára akkor lehet szükség, ha a kódban számos példányosító utasítás ismétlődik, és célunk, hogy ezt a folyamatot egyszerűsítsük.

Az Abstract Factory célja, hogy kapcsolódó vagy függő objektumcsaládokat hozzon létre anélkül, hogy a konkrét osztályokat megneveznénk. Segítségével egy közös témához tartozó, egyedi gyártó metódusokat csoportosíthatunk anélkül, hogy specifikálnánk azok konkrét osztályait.

Ez a minta akkor hasznos, ha a rendszer független a termékek szerkezetétől és gyártási folyamatától, több termékcsaládot kell kezelni, és a termékcsalád tagjait

együtt kell használni. Emellett a termékek implementációja rejtett, csak az interfész ismert.

## Szerkezeti minta:

A szerkezeti minták célja az, hogy a különböző objektumokat és osztályokat hogyan lehet hatékonyan összekapcsolni úgy, hogy azok jól együttműködjenek, miközben fenntartják a kód olvashatóságát és egyszerűségét.

Az adapter tervezési mintát akkor alkalmazzuk, ha egy adott kód interfészét egy olyan interfésszel szeretnénk kompatibilissé tenni, amit a kliens már ismer. Erre általában akkor van szükség, ha egy meglévő rendszert szeretnénk egy másikhoz hozzáigazítani. Motivációs példa lehet, ha például egy grafikus szerkesztőben szeretnénk használni egy már meglévő könyvtárat vagy modult, amelyet egy szövegszerkesztőben használtak, de alapból nem alkalmazható a grafikus szerkesztőben, mivel az interfésze nem megfelelő.

A **Composite** minta célja, hogy a rész-egész struktúrákat objektum-hierarchiaként (fa-szerkezetként) ábrázolja, miközben a kliens a rész- és egész elemeket ugyanúgy kezeli.

Motivációs példa lehet egy rajzszerkesztő, amelyben primitív elemekből építünk fel összetett elemeket, akár rekurzívan is. Ilyenkor célunk, hogy az összetett elemeket ugyanúgy kezelhessük, mint a rész elemeket (és fordítva). Más szóval, függetlenül attól, hogy összetett vagy primitív elemmel dolgozunk, ugyanazokkal a viselkedésekkel láthatjuk el őket. Az összetett elemek annyiban különböznek, hogy képesek további elemeket hozzáadni vagy eltávolítani, viselkedésüket pedig úgy definiálhatjuk, hogy azokat a hozzájuk tartozó gyermekelemekre alkalmazzuk. A lényeg, hogy a kliensnek nem kell különbséget tennie a primitív és az összetett elemek között; mindkettőt ugyanúgy tudja kezelni.

A **Decorator** tervezési minta célja, hogy új felelősségeket vagy tulajdonságokat dinamikusan csatoljunk egy objektumhoz. Motivációs példa lehet, ha további funkcionalitást szeretnénk rendelni egyes objektumokhoz, de nem az összeshez, így nem akarjuk az egész osztályt módosítani. Bár ezt megoldhatnánk öröklődéssel is, futásidőben az objektum típusát nem változtathatjuk, míg a különböző dekorátorokat könnyen cserélhetjük. A dekorálás során egy dekoráló objektumba csomagoljuk az eredeti objektumot, és új funkcionalitásokat adunk hozzá, miközben az eredeti objektum viselkedését elérhetővé tesszük ezen keresztül. Ez a minta minden olyan helyzetben alkalmazható, amikor a származtatás nem lehetséges vagy nem praktikus.

## Viselkedési minták

A viselkedési minták a tervezési minták egyik alapvető kategóriáját alkotják, amelyek arra összpontosítanak, hogy hogyan kommunikálnak és együttműködnek az objektumok a rendszerben. A viselkedési minták célja, hogy biztosítsák az objektumok közötti megfelelő interakciót anélkül, hogy túlzottan összetett lenne a kód. Az ilyen típusú minták segítenek a programok logikai folyamatainak

kezelésében, az objektumok közötti kommunikációs csatornák kialakításában, és a rendszer viselkedésének hatékony irányításában.

Az **Observer minta** akkor hasznos, ha egy objektum állapotának változását több más objektum is figyelemmel kíséri, és ezeket a figyelőket értesíteni kell, amikor az állapot változik. A minta lehetővé teszi, hogy az egyik objektum (a "subject" vagy "tárgy") értesítse az összes kapcsolódó objektumot (az "observer"-eket), ha annak állapota módosul. Például GUI alkalmazásoknál alkalmazható, ahol a felhasználói felület különböző részei reagálnak a háttérben történő változásokra.

A **Strategy minta** lehetővé teszi, hogy egy algoritmust dinamikusan válasszunk ki egy objektum futásidejében. Ez a minta elválasztja az algoritmusokat a használatuktól, és lehetővé teszi, hogy azokat könnyen cseréljük anélkül, hogy az alkalmazás többi részét módosítani kellene. A minta segít az algoritmusok rugalmassá tételében. Például ha különböző algoritmusok cserélgetünk anélkül, hogy az egész rendszert újra kellene tervezni vagy h a a rendszerben sokféle műveletet kell végrehajtani, és azokat dinamikusan kell váltogatni.

A **Command minta** lehetővé teszi, hogy egy kérést (vagy parancsot) objektumként kezeljünk. Ezáltal a kéréseket és azok végrehajtásait különböző objektumoknak átadhatjuk, késleltethetjük vagy visszavonhatjuk. A minta segít az utasítások, parancsok, vagy műveletek kezelése során a megfelelő logika elkülönítésében.

Az **Iterator minta** lehetővé teszi, hogy egy objektum gyűjteményén (pl. lista, halmaz) iteráljunk anélkül, hogy az iterálás részleteit a kliens kódjában kellene kezelni. A minta biztosítja, hogy egy gyűjtemény elemeit egységes módon bejárhassuk, függetlenül attól, hogy milyen típusú gyűjteményről van szó.

Egy könyvtárban különböző típusú könyveket tartunk nyilván, és szeretnénk ezeket sorra végignézni. Az Iterator minta biztosítja, hogy ugyanazt az iterációs mechanizmust alkalmazhassuk, függetlenül attól, hogy a könyvek listában, halmazban vagy más típusú gyűjteményben vannak-e tárolva. Például adatgyűjtemények kezelése, amelyeknek van egyértelmű és konzisztens módja az elemek végigjárásának vagy ha egy rendszer többféle adatstruktúrát használ, és egységes módon akarjuk bejárni azokat.

A **Mediator minta** közvetítő objektumot használ a kommunikációs kapcsolatok kezelésére. Ezzel elkerülhetjük, hogy az objektumok közvetlenül kommunikáljanak egymással, csökkentve a rendszer komplexitását. A mediátor közvetíti a különböző objektumok közötti interakciókat.

Például egy chat alkalmazásban több felhasználó is beszélgethet egymással. A Mediátor minta használatával nem minden felhasználónak kellene tudnia minden más felhasználóról; egy központi "mediátor" objektum kezeli a kommunikációt, és irányítja az üzeneteket a megfelelő személyekhez.