Tervezési minták egy OO programozási nyelvben

Duncsák Mihály
neptun kód: R86N3I

Törzsszám: LB_PM 1420/2023
Képzés kódja: LB_PM
Képzés neve: Programtervező informatikus
Szint: alapképzés (BA/BSc/BProf)
Tagozat: Levelező

2024

Tartalomjegyzék

Bevezetés

Az **objektum-orientált programozás (OOP)** egy programozási paradigma, amely az **objektumok** köré szerveződik. Az objektumok olyan egységek, amelyek tartalmazzák a program adatait (**attribútumok**) és az azokon végzett műveleteket (**metódusok**). Az OOP célja, hogy a szoftverfejlesztés struktúráját a valós világ problémáira jobban hasonlító módon modellezze, ezzel növelve a programok érthetőségét, rugalmasságát és újra felhasználhatóságát.

Fő koncepciók:

Osztály (Class):

Az osztály egy sablon vagy tervrajz, amely alapján objektumokat hozunk létre.

Meghatározza az objektum tulajdonságait (attribútumok) és viselkedését (metódusok).

Példa: Egy *"Autó"* osztály leírhatja az autó tulajdonságait, mint szín, sebesség, és metódusokat, mint gyorsítás vagy fékezés.

Objektum (Object):

Az objektum egy konkrét példány az osztályból.

Példa: Egy adott autó, amely kék színű és 120 km/h sebességgel halad.

Egy programban több objektum is létezhet ugyanabból az osztályból.

Encapsulation (Adatrejtés):

Az adatok és a logika egyetlen egységbe zárása, hogy azok ne legyenek közvetlenül elérhetők az osztályon kívülről.

Az osztály csak a szükséges metódusokon keresztül engedi a hozzáférést az adataihoz, ezzel biztosítva a biztonságot és a modularitást.

Példa: Egy banki számla osztály csak meghatározott metódusokon keresztül engedheti a pénz hozzáadását vagy levételét.

Inheritance (Öröklődés):

Az öröklődés lehetővé teszi, hogy egy osztály (alaposztály) tulajdonságait és metódusait egy másik osztály (gyermekosztály) megörökölje.

Ez újra felhasználhatóvá és könnyen bővíthetővé teszi a kódot.

Példa: Egy "Jármű" osztály általános tulajdonságait (sebesség, üzemanyag) a "Motor" és "Kerékpár" osztályok öröklik.

Polymorphism (Polimorfizmus):

Az az elv, amely lehetővé teszi, hogy különböző osztályokhoz tartozó objektumok ugyanarra a metódusra különböző módon reagáljanak.

Ez dinamikus viselkedést tesz lehetővé az alkalmazásban.

Példa: Egy "Rajzol" metódus különbözően működhet egy "Kör" vagy egy "Négyzet" osztály esetében.

Abstraction (Absztrakció):

Az absztrakció elrejti az objektum belső működésének részleteit, és csak a lényeges információkat mutatja meg a külvilág számára.

Ez segíti a komplex rendszerek egyszerűbb kezelhetőségét.

Példa: Egy autó használatakor nem kell tudnunk, hogyan működik a motor, csak azt, hogy a gázpedált kell nyomnunk a gyorsításhoz.

Az OOP előnyei

Modularitás:

A program részei (osztályok) egymástól függetlenül fejleszthetők és tesztelhetők.

Újra felhasználhatóság:

Az osztályok és objektumok újra felhasználhatók más projektekben vagy ugyanazon projekten belül.

Rugalmasság:

Az OOP könnyen bővíthető új funkcionalitásokkal anélkül, hogy jelentős változtatásokat kellene végezni a meglévő kódon.

Karbantarthatóság:

Az OOP alapelvei alapján írt kód strukturáltabb, így könnyebb javítani és karbantartani.

Példák az objektum-orientált nyelvekre:

Java, **C++**, **C#**, **Python**, **Ruby**

Tervezési minták egy OO programozási nyelvben

1. MVC (Model-View-Controller) Minta

Az **MVC** (Model-View-Controller) minta az egyik legismertebb és legelterjedtebb tervezési minta a szoftverfejlesztésben, különösen az olyan alkalmazások esetében, amelyek felhasználói felületet (UI) tartalmaznak. Az MVC minta alapja **a szétválasztott felelősségek koncepciója**, amely lehetővé teszi a kód tisztán tartását és a könnyebb karbantartást.

Három fő összetevője:

- **Model (Modell)**: A modell a program adatainak és logikájának a reprezentációja. A modell felelős az adatok tárolásáért és azok manipulálásáért, és nem tartalmaz semmilyen információt a felhasználói felületről. A modellhez tartozik minden üzleti logika és adatkezelés.
- **View (Nézet)**: A nézet felelős a felhasználói felület (UI) megjelenítéséért. A nézet figyeli a modellt, és az adatokat a felhasználó számára megfelelő módon jeleníti meg. A nézet nem tartalmaz üzleti logikát; csak az adatokat formázza és jeleníti meg a felhasználó számára.
- **Controller (Vezérlő)**: A vezérlő szerepe az, hogy összekapcsolja a modellt és a nézetet. A vezérlő fogadja a felhasználói interakciókat (például gombnyomásokat, űrlapok beküldését), és a megfelelő modellhez vagy nézethez irányítja őket. A vezérlő tehát az adatkezelés és a megjelenítés közötti hidat képezi.

Példa alkalmazásban: Egy webalkalmazásban, ahol a felhasználó adatokat küld egy űrlapon, az MVC minta segít elválasztani a felhasználói interakciókat (controller), az adatkezelést (model) és az adatokat megjelenítő felhasználói felületet (view).

Előnyök:

- Kód tisztábbá válik, mivel a logikai elemek és a UI elemek külön vannak választva.
- Könnyebben módosíthatóak az egyes részek a másik módosítása nélkül.
- Tesztelhetőség javul: a logika külön van választva a nézettől, így az üzleti logika könnyebben tesztelhető.

2. Singleton Minta

A **Singleton** minta biztosítja, hogy egy osztályból csak egyetlen példány létezzen, és ez az egyetlen példány globálisan elérhető legyen. Ha valamilyen erőforrást, például adatbázis-kapcsolatot, konfigurációs beállításokat vagy logikai egységet szeretnénk, akkor a Singleton minta segíthet, hogy egyetlen példány legyen az alkalmazásban.

Jellemzők:

- Az osztály maga biztosítja a példány létrehozásának módját, biztosítva, hogy csak egy példány létezzen.
- A globális példány elérhetősége biztosítva van.
- Használható olyan esetekben, amikor az erőforrások megosztása szükséges.

Előnyök:

- Csökkenti az erőforrások túlzott létrehozását (például adatbázis-kapcsolatokat).
- Globálisan hozzáférhetővé teszi az egyetlen példányt.

3. Observer Minta

Az **Observer** minta lehetővé teszi, hogy egy objektum (az úgynevezett "subject") értesítse az összes hozzá kapcsolódó objektumot (az "observer"-eket), ha az állapota megváltozik. Ez különösen hasznos, amikor egy objektum állapotának változása több más objektumot is érinthet.

Jellemzők:

- A subject tárolja az összes observer-t és értesíti őket, ha változik az állapotuk.
- Az observer-ek reagálnak az állapotváltozásokra.

Példa: Egy grafikus felhasználói felületen a felhasználó egy gombot nyom meg (subject), amely számos más komponens (observer) viselkedését módosíthatja (például frissíti az adatokat).

Előnyök:

- Könnyen hozzáadhatunk új observer-eket anélkül, hogy meg kellene változtatnunk a subject-et.

- Elválasztja az értesítést a reakciótól.

4. Factory Minta

A **Factory** minta célja, hogy objektumokat hozzon létre anélkül, hogy a kliensnek meg kellene ismernie az objektumok pontos típusát. Ez különösen hasznos, amikor az objektumok létrehozása összetett vagy változó típusú, és a rendszer többi részét el akarjuk vonni a konkrét típusok kezelésétől.

Jellemzők:

- A gyár osztály felelős az objektumok létrehozásáért.
- A kliens kód nem kell, hogy tudja, milyen típusú objektumokat hoz létre.

Példa: Egy alkalmazásban, ahol különböző típusú alakzatokat (kör, négyzet, háromszög) hozunk létre, egy Factory minta segíthet az alakzatok típusának eldöntésében a háttérben.

Előnyök:

- Kód tisztábbá válik, mivel a létrehozás részletei el vannak választva a fő logikától.
- Könnyen bővíthető, új típusok hozzáadásával nem kell változtatni a klienseken.

5. Strategy Minta

A **Strategy** minta lehetővé teszi, hogy egy algoritmus családját különböző implementációkkal biztosítsuk, és az algoritmusokat dinamikusan cseréljük az alkalmazás futása közben. Ezzel a mintával az algoritmusok változtathatók anélkül, hogy a kód többi részét módosítani kellene.

Jellemzők:

- Az algoritmusokat különböző osztályokba helyezzük, és azokat különböző módon használjuk.
- A kliens dönthet, hogy melyik algoritmust szeretné használni.

Példa: Egy szállítási alkalmazásban különböző szállítási módokat (autó, hajó, repülő) kezelünk, és dinamikusan választhatjuk ki, hogy melyiket alkalmazzuk a rendelés szállításához.

Előnyök:

- Az algoritmusok változtathatók anélkül, hogy az alkalmazás többi részét módosítani kellene.
- Az alkalmazás rugalmassá válik.

6. Decorator Minta

A **Decorator** minta lehetővé teszi, hogy egy objektumot dinamikusan bővítsünk új funkciókkal anélkül, hogy öröklődést kellene alkalmaznunk. A dekorátorok egyesíthetik az eredeti objektum funkcióit további viselkedésekkel, és a viselkedésük futás közben változtatható.

Előnyök:

- Rugalmasan bővíthetjük a funkciókat anélkül, hogy az osztályokat módosítani kellene.
- Különböző dekorátorok kombinálásával többféle viselkedést érhetünk el.

7. Adapter Minta

Az **Adapter** minta lehetővé teszi, hogy két inkompatibilis interfész között kapcsolatot hozzunk létre. Az adapter átalakítja az egyik osztály interfészét, hogy az a másik osztály számára is érthető legyen.

Előnyök:

- Az inkompatibilis rendszerek közötti kommunikációt megkönnyíti.
- Új rendszerek integrálása könnyen megoldható.

8. Command Minta

A **Command** minta lehetővé teszi, hogy a felhasználói interakciókat, mint például gombnyomásokat, parancsok formájában tároljuk, és késleltetve vagy újra végrehajtsuk őket. A minta segíthet a felhasználói műveletek dekódolásában és visszavonásában.

Előnyök:

- Parancsok késleltetése és újrafuttatása egyszerű.
- Az operációk egyszerű nyilvántartása és visszavonása.
-

Az objektum-orientált programozás lényege az adatok és a viselkedésük egyesítése, az osztályok és objektumok segítségével. E tervezési minták az OO alapelvekre építenek, mint például az **öröklődés**, **polimorfizmus**, **absztrakció** és **kapcsolódás csökkentése** (low coupling):

- Az MVC minta az objektum-orientált programozásra épít, mivel három különböző osztályt (model, view, controller) hoz létre, amelyek mindegyike különböző felelősséggel bír. Az osztályok közötti interakciókat és kommunikációt is szabályozza. A vezérlő például felelős a modellekhez való hozzáférésért, míg a nézet kizárólag a megjelenítésért.
- Az OO-ban az osztályok példányosítása általában dinamikusan történik. A Singleton minta egyedülálló példányokat biztosít egy osztályból, biztosítva, hogy csak egy példány létezzen. Az osztályokon keresztüli kontrollált példányosítás OO szemléletben jelenik meg.
- Az Observer minta az **event-driven** (eseményalapú) programozásra épít, amely az objektumok közötti interakciókat és kapcsolatokat használja. Az **osztályok** (subject és observer) közötti kapcsolatot az OO polimorfizmusával és öröklődésével könnyen implementálhatjuk.
- A Factory minta szoros kapcsolatban áll az **abstrakcióval** és az **összetett példányosítással**. Az objektumok létrehozásához szükséges részletek elrejtése az osztályokban az OO alapelvei szerint történik, például a *deklaratív interfészek* alkalmazásával, ahol az implementációk eltérhetnek, de az interfész ugyanaz marad.
- Az Strategy minta az **öröklődés** és **polimorfizmus** alapjaira épít. A különböző algoritmusokat osztályokba helyezhetjük, és azokat dinamikusan válthatjuk ki az adott környezetben, ezzel biztosítva az alkalmazás rugalmasságát. Az osztályok közötti cserélhetőség az OO alapelvek kihasználásával valósul meg.

- A Decorator minta a **kompozíció** és **dizájn minta** alapú OO megközelítést alkalmaz. Az objektumokat új funkcionalitással bővítjük ahelyett, hogy örökléssel módosítanánk őket, így elkerülve az öröklődési hierarchiák túlbonyolítását.
- Az Adapter minta az OO-ban az **interfész-ek** kezelésére épít, mivel az inkompatibilis osztályok közötti kommunikációt biztosítja. Az adapter egy olyan osztályt hoz létre, amely az egyik interfészhez való hozzáférést átalakítja egy másik interfészhez, miközben mindkét oldalon megőrzi az objektumok tisztaságát.
- A Command minta az **abstrakció** és az **interakciók kezelésére** épít, mivel a felhasználói műveleteket objektumokká alakítja, így azok könnyen kezelhetők, késleltethetők és visszavonhatók. Az osztályok felelősségi körét tisztán elválasztja az alkalmazás logikájától, elősegítve a kód rugalmasságát.
-

Ezek a minták az **osztályok** és **objektumok közötti felelősség megosztására** épít, valamint az osztályok **dinamikus** és **flexibilis használatát** célozza meg. A legtöbb minta az OO alapelveket, mint az öröklődés, polimorfizmus, és enkapszuláció használja, hogy jobb, könnyebben karbantartható, és bővíthető kódot készíthessünk.

Összefoglalás

Mivel az objektum-orientált programozás a valós világ modellezésére törekszik, ezek a minták segítenek a fejlesztőknek abban, hogy a kódot jobban szervezetté, tisztábbá és karbantarthatóbbá tegyék.