Tervezési minták egy OO programozási nyelvben.

MVC, mint modell-nézet-vezérlő minta és néhány másik tervezési minta

Tervezési minták OO programozási nyelvekben

A tervezési minták általános megoldások a szoftverfejlesztésben, azon belül is a szoftver dizájnnál előforduló gyakori problémákra. Mindegyik tervezési minta gyakorlatilag egy tervrajz, ami szabadon, az adott probléma megoldására szabható, alakítható. Ezekkel a mintákkal gyakran előforduló design problémákat oldhatunk meg rövid idő alatt.

A tervezési minták nem konkrét kódok, kódrészletek, vagy library-k, sokkal inkább koncepciók egy bizonyos probléma megoldására. Ugyancsak nem összekeverendő az algoritmusokkal, ahol mindig van egy jól, és pontosan meghatározott lépések halmaza a cél eléréséhez. A minták inkább egy sokkal magasabb szintű leírásai egy adott problémának.

A legtöbb tervezési minta egységes leírással rendelkezik, hogy a fejlesztők képesek legyenek sok, különböző kontextusban is reprodukálni a mintákat. Általában a minták a következő elemeket tartalmazzák:

- **Cél:** A minta célja általában tartalmazza röviden a megoldani kívánt problémát, és a hozzá tartozó megoldást is.
- **Motiváció:** Részletesebb kibontása a célnak, mind a probléma, mind pedig a megoldás esetében.
- **Struktúra:** Az osztályok struktúrája, itt meg lehet mutatni a minta összes elemét, és hogy azok hogy kapcsolódnak egymáshoz.
- **Példakód:** Általában a gyakran, sokak által használt programozási nyelvek egyikén írt minta kód, ami bemutatja a minta működését.

A tervezési minták lehet alacsony szintűek, amik jellemzően egy adott programozási nyelvhez használhatóak, vagy magas szintűek, amik a teljes szoftver felépítéséhez (architecture) köthetők, és gyakorlatilag bármilyen nyelven lefejleszthetők.

A leggyakoribb csoportosítás cél szerint történik, a legfőbb csoportok a következők:

- **Creational:** Objektumok létrehozásának mintái, a segítségével rugalmasabb működést érhetünk el, és fokozhatjuk a meglévő kódot újra felhasználásának lehetőségét.
- **Structural:** Megmutatja, hogy hogyan lehet az objektumokat és osztályokat nagyobb struktúrába szervezni, miközben maga a struktúra rugalmas és hatékony marad.
- **Behavioral:** A hatékony kommunikáció, és az objektumok közötti felelősségi körök pontos meghatározása a fő célja.

A tervezési minták kipróbált, és letesztelt megoldások különböző, a szoftverfejlesztési folyamat során előforduló problémákra. Még ha nem is találkozik a fejlesztő ezekkel a problémákkal, akkor is érdemes tudni róluk, mert azt tanítják meg, hogy hogyan lehet bármilyen problémát az objektum-orientált programozási módszerek segítségével megoldani. Ezen kívül egy úgymond közös nyelv is lehet a fejlesztői csapaton belül, hiszen elég csak névvel hivatkozni egy mintára, és máris lehet tudni, hogy az adott feladatot hogy lehet a leghatékonyabban megoldani.

MVC – Modell-Nézet-Vezérlő minta

Ez a minta leginkább architectural mintának mondható, viszont az a sajátossága, hogy nem a teljes applikációt írja le. Leginkább csak a felhasználói felület, illetve az interakciós rétegek létrehozásához kapcsolódik. Ezen kívül még számos réteg szükséges egy applikációhoz, mint például az üzleti logika réteg, a sevice réteg, vagy adat hozzáférési réteg.

Az MVC lényege, hogy meghatározza, hogy az alkalmazásnak tartalmaznia kell egy adat modellt, és prezentációs réteget, ami megjeleníti az adatokat, és egy vezérlőt, ami a kapcsolatot biztosítja a modell, és prezentációs réteg között, illetve kezeli az interakciókat. A minta szerint ezeknek a feladatköröknek mind külön objektumként kell szerepelniük az applikációban.

Modell: Ez tisztán az alkalmazás adatai, és annak a logikája, hogy hogyan lesz az adat prezentálva a felhasználó felé. Ez általában egy sima, egyszerű Java osztály szokott lenni, konstruktorral, és getter-setter metódusokkal.

Nézet: Ez az objektum megjeleníti a model adatait (nem az alkalmazás adatai), tehát itt láthatóak azok az adatok, amiket a forrásból, például egy SQL szerverről beolvastunk. A nézet tudja, hogy hogy érje el az adatokat, viszont nem tudja, hogy mik az adatok amiket megjelenít, és azt sem tudja, hogy a felhasználónak milyen lehetőségei vannak arra, hogy az adatokat manipulálja. Összefoglalva, a nézet csupán megjelenít, reprezentál. Java esetén általában ez egy HTML vagy JSP file.

Vezérlő: A vezérlő a nézet és a modell között helyezkedik el. A legfontosabb feladata, hogy figyeli és feldolgozza a nézetből (vagy más külső forrásból) érkező kéréseket, és a kérésnek megfelelő reakciót indít el. Általában ez a reakció nem más, mint egy metódus hívás a modellen. Mivel a nézet és a modell egy értesítésen alapuló kapcsolatban van egymással, így a változtatások azonnal megjelennek a nézetben is.

Előnyei:

- Egyszerre több fejlesztő is dolgozhat egy időben a modellen, a vezérlőn és a nézeten.
- Az MVC modell lehetővé teszi az összetartozó metódusok logikai csoportosítását a vezérlőben. Ugyanígy, ha több nézetre van szükség, azok is ugyanígy csoportosíthatók
- Egy modellnek több nézete is lehet.
- Az alkalmazás komponensei könnyen kezelhetők, és kevésbé függenek egymástól.

Hátrányok:

- Mivel több réteget, absztrakciót is készítünk, így a navigáció az alkalmazás komponensei között bonyolulttá válhat.

Néhány további tervezési minta

Factory method (Virtuális konstruktor):

Ez egy creational minta, ami egy interface-t biztosít az objektumok létrehozásához egy úgynevezett szuperosztályban, viszont megengedi az alosztályoknak, hogy módosíthassanak a létrehozandó objektumtípusokon.

Ez a minta azt javasolja, hogy változtassuk meg a direkt objektumlétrehozást a new operátorral, és inkább egy speciális Factory metódust használjunk. Az objektumok így is a new operátorral fognak készülni, de a factory metódusból lesz meghívva. Így a konstruktor egy központi helyen lesz, viszont az alosztályok módosíthatják saját hatáskörükben, hogy milyen objektumot hoznak létre.

Adapter (Wrapper):

Ez egy strukturális minta, ami megmutatja, hogy hogyan tudnak a nem kompatibilis interface-ekkel rendelkező objektumok együttműködni.

Az ehhez tartozó probléma leggyakrabban a különböző formátumú adatok kezelésénél jön elő, például ha az egyik rendszerünk JSON formátumban teszi elérhetővé az adatait, de nekünk XML formátumra van szükségünk. Ilyenkor nem a legegyszerűbb dolog valamelyik library-t, vagy a már meglévő kódbázist átírni, hanem létre kell hozni egy adaptert, ami átkonvertálja az adatot az egyik formátumból a másikba.

Template:

Ez egy behavioral design minta, ami definiál egy algoritmus alapszerkezetet egy szuperosztályban, de közben engedélyezi, hogy az alosztályok felülírhassanak bizonyos lépéseket az algoritmusban, de anélkül, hogy megváltoztathatnák a struktúrát.

A legjobb példa erre egy olyan funkció, ami különböző file formátumokból tud beolvasni és elemezni adatokat. Az algoritmus alapvetően ugyanazon a folyamaton megy át minden formátum esetében, ilyen a file beolvasás, betöltés, elemzés, eredmények mentése, bezárás, stb. Amíg például az elemzés teljesen ugyanúgy zajlik, a megnyitáshoz például már más kell egy .doc file esetében, mint egy .xlsx-nél. Ilyenkor például a Word dokumentummal foglalkozó alosztály felülírhatja az openFile(), vagy extractData() metódust, és ugyanígy tehet az Excel dokumentumot kezelő alosztály is. Viszont az analyzeData() metódus már ugyanazt csinálja mindkét esetben, ezért ez maradhat a szuperosztályból származtatott metódus, így nincs szükség a kód duplikációra sem.