

Gang-of-Four tervezési minták 3

Singleton (Creational pattern)

- Biztosítja, hogy egy osztályból csak egy példányt lehessen létrehozni és ehhez az egy példányhoz globális hozzáférést biztosít.
- **Probléma**
 - o Van egy objektumunk és egy idő után feltűnik, hogy ugyanazt az objektumot használtuk.
 - o Globális változók lehet tárolnak fontos dolgokat, de mégis felül lehet írni kívülről.
- **Megoldás**
 - o Legyen az osztály felelőssége, hogy csak egy példányt lehessen belőle létrehozni.
 - o Biztosítson hozzáférést ehhez az egy példányhoz.
 - o Az Instance osztály-művelet (statikus) meghívásával lehet példányt létrehozni, illetve az egyetlen példányt elérni.
- **Az Instance**
 - o Mindig ugyanazt az objektumot adja vissza.
 - o C# esetén property-vel célszerű: Singleton.Instance
- A Singleton konstruktora protected láthatóságú.
 - o Ez garantálja, hogy csak a statikus Instance metódushíváson keresztül lehessen példányt létrehozni.

Singleton használati esetek

- Ha egy osztálynak csak egyetlen példánya kell, hogy legyen, ami minden kliens számára elérhető. (Pl.: egyetlen adatbázis-objektum, amit a program különböző részei megosztanak.)
- Szigorúbb ellenőrzésre van szüksége a globális változók felett.

Singleton implementálása

1. Privát statikus mező létrehozása az osztályban a singleton példány tárolására.
2. Nyilvános statikus létrehozási metódus deklarálása a singleton példány kinyeréséhez.
3. A statikus metóduson belül inicializálás végrehajtása.
 - a. Első híváskor az új objektum létrehozása és statikus mezőbe helyezése.
 - b. A metódusnak minden további híváskor mindig ezt a példányt kell visszaadnia.
4. Az osztály konstruktora legyen privát.
 - a. Az osztály statikus metódusa továbbra is képes lesz meghívni a konstruktort, de a többi objektum nem.

Singleton előnyei és hátrányai

- **Előnyei**
 - o Egyetlen példánya van az osztálynak, globális pontot biztosít ehhez a példányhoz.
 - o A singleton objektum csak akkor inicializálódik, amikor először kérjük.
- **Hátrányai**
 - o Speciális kezelést igényel többszörös környezetben, hogy több szál ne hozzon létre többször egy singleton objektumot.
 - o Nehezíti a Unit tesztelést, mock objektum előállítását nehézkes. Konstruktort privát.

Prototype (Creational pattern)

- A prototípus alapján új objektumpéldányok készítése.
- Minden objektum támogatja (Object osztály művelete)
 - o Shallow copy
- Igazi, publikus, mély másolatot végző klónozáshoz implementálható az ICloneable interfész
 - o Deep copy
- **Probléma**
 - o Át akarunk másolni minden egyes objektumot, de lehetnek olyan mezők, amik privátok, nem láthatóak kívülről.
 - o Másik probléma, hogy mivel a duplikátum létrehozásához ismerni kell az objektum osztályát, a kód függővé válik az osztálytól.
- **Megoldás**
 - o Létrehozunk egy interfészt, ami az összes objektumnak elérhető.
 - o Ezáltal lehet klónozni, ami egy Clone metódus.
 - o A metódus létrehoz egy objektumot az aktuális osztályból és a régi objektum összes mezőértékét átviszi az új objektumba. (Így már a privát mezők is másolhatóak.)
 - o **Azaz objektum, ami támogatja a klónozást, azt hívjuk prototype-nak.**

Prototype használjuk, ha

- Egy rendszernek függetlennek kell lennie a létrehozandó objektumok típusától.
- Ha a példányosítandó osztályok futási időben határozhatók meg.
- Ha nem akarunk nagy párhuzamos osztályhierarchiákat.
- Amikor az objektumok felparaméterezése körülményes és könnyebb egy prototípust inicializálni, majd azt átmásolni.

Prototype implementálása

1. Prototype interfész létrehozása, amiben van egy Clone metódus vagy interfész nélkül.
2. A prototype osztálynak lennie kell egy alternatív konstruktornak, ami elfogadja az adott osztály egy objektumát.
 - a. A konstruktornak az átadott objektumból az osztályban definiált összes mező értékét át kell másolnia az újonnan létrehozott példányba.
 - b. Ha egy alosztályt változtatunk, akkor meg kell hívunk a szülő konstruktort, hogy az őosztályt kezelje a privát mezők klónozását.
3. Clone metódus felülírása new operátorral, ezáltal új logikát adhatunk neki.

Prototype előnyök és hátrányok

- **Előnyök**
 - o Objektumok hozzáadása és elvétele futási időben
 - o Új, változó struktúrájú objektumok létrehozása
 - o Redukált származtatás, kevesebb alosztály
- **Hátrányok**
 - o Minden egyes prototípusnak implementálnia kell a Clone() függvényt, ami bonyolult lehet.

Builder (Creational pattern)

- Lehetővé teszi az összetett objektumok lépésről-lépésre történő létrehozását.
- **Probléma**
 - o Van egy összetett objektum, ami számos mezőt és egymásba ágyazott objektumot tartalmaz, ami inicializálást igényel.
 - o Egy ilyen inicializálási kód általában sok paramétert tartalmazó konstruktorban van elrejtve vagy még rosszabb, ha a kliens kódban vannak szétszórva.
 - o **Túl bonyolulttá teheti a programot, ha egy objektum minden lehetséges konfigurációjára létrehoz egy alosztályt.**
 - o **Túl sok paramétere van a konstruktornak, ez így nagyon csúnya.** (Lehet a paraméterek egy része nem is kell.)
- **Megoldás**
 - o Az objektum létrehozásának kódját ne a saját osztályába rakjuk bele, hanem helyezzük át egy builder objektumba.

Builder használati esetek

- Telescoping konstruktoroktól mentesség (pl.: Egy konstruktor egy paraméternek, másik konstruktor másik paraméternek, stb.)
- Objektum felépítése lépésről-lépésre.

Builder implementálása

1. Határozzuk meg a builder lépéseit. (Pl.: Hogyan építsünk fel egy objektumot)
2. Base builder interfész kialakítása.
3. Builder osztály létrehozása, ami implementálja a builder interfészt.
4. Director osztály létrehozása.
 - a. Különböző metódusokat tartalmazhat az objektumok létrehozására.
5. Kliens kód használja a builder és a director objektumokat.
 - a. Először a builder objektumot át kell adni a director-nak konstruktoron keresztül paraméterként.
 - b. Innentől kezdve a director használja a builder-t.
6. Builder eredmény akkor születik a director-ból, ha minden elem ugyanazt az interfészt használja.
 - a. Ellenkező esetben a kliensnek az eredményt a builder-től kell lekérnie.

Builder előnyök és hátrányok

- **Előnyök**
 - o Lépésről-lépésre való „építkezés”/building.
 - o Single Responsibility elv-et követi.
 - o Komplex kód elkülönítése a business logic-tól.
- **Hátrányok**
 - o A kód komplexitása növekszik, mivel több új osztály létrehozását igényli.

Iterator

- Szekvenciális hozzáférést egy összetett (pl.: lista) objektum elemeihez anélkül, hogy annak belső reprezentációját felfedné.

- **Probléma**
 - Legyen bármilyen gyűjteményünk ezeket szeretnénk egy bejárható interfészen keresztül elérni.
- **Megoldás**
 - Adatszerkezeten implementáljuk az `IEnumerable<T>` és egy külső bejáró osztályon az `IEnumerator<T>` interfészt. Előírja ezeket a metódusokat:
 - **`void Reset()`**: Gyűjtemény elejére visszaállítás
 - **`bool MoveNext()`**: Következő elemre lépés
 - **`T Current`**: Visszaadja az aktuális elemet.

Iterator használjuk, ha

- Úgy szeretnénk hozzáférni egy objektum tartalmazott objektumaihoz, hogy nem akarjuk felfedni a belső működését.
- Többféle hozzáférést szeretnénk biztosítani a tartalmazott objektumokhoz.
- Egy időben több, egymástól független hozzáférést szeretnénk a lista elemeihez.
- Különböző tartalmazott struktúrákhoz szeretnénk hozzáférni hasonló interfésszel.

Iterator előnyök és hátrányok

- **Előnyök**
 - Single Responsibility elv és Open/Closed elv
- **Hátrányok**
 - Iterator nem biztos, hogy olyan hatékony, mint egyes gyűjtemények elemeinek közvetlen végigjárása.

Chain of Responsibility (Behavioral pattern)

- Az üzenet vagy kérés küldőjét függetleníti a kezelő objektum(ok)tól.
- **Probléma**
 - Validálások szekvenciálisan hajtódnak végre.
 - Egy idő után bonyolulttá, átláthatatlanná válik a kód a sok validáció miatt.
 - Esetleges duplikációk is felmerülhetnek, mert lehet kell egy másik validáció miatt.
- **Megoldás**
 - Önálló objektumokká, handlerekké (handlers) alakítunk át.
 - Egy ellenőrzés egy metódus, paraméterként adunk át adatokat.
 - A handlereket láncba kell kapcsolni.
 - Minden összekapcsolt handler rendelkezik egy mezővel, ami a lánc következő handlerre való hivatkozást tárolja.
 - A kérés feldolgozása mellett a handlerek továbbítják a kérést a láncban.
 - A kérés addig halad a láncban, amíg az összes handler fel nem tudja dolgozni azt.
 - A handler dönthet úgy, hogy nem továbbítja a kérést a láncba és ezzel leállítja a további feldolgozást.

Chain of Responsibility implementálása

1. Handler interfész deklarálása és a kérések kezelésére szolgáló metódus leírása.

2. A handlerekben található kódok kiküszöbölése érdekében érdemes létrehozni egy absztrakt alap kezelő osztályt, ami a handler interfészből származik.
 - a. Kell egy mező, ami rámutat a következő handler-re.
 - b. Ha a láncokat futásidőben módosítani akarjuk, akkor a referencia mező értékének megváltoztatására egy setter-t kell definiálni.
3. Egyenként hozzunk létre handler alosztályokat és a handler metódusokat valósítsuk meg.
 - a. Minden handler-nek két döntést kell hoznia, amikor egy kérést fogad:
 - i. Feldolgozza-e a kérést vagy továbbítja-e a kérést a láncban.
4. A kliens saját maga állíthat össze láncokat vagy más objektumoktól kaphat előre elkészített láncokat.
5. A kliens bármelyik handlert elindíthatja.
 - a. A kérés addig halad végig a láncon, amíg valamelyik kezelő el nem utasítja a továbbküldést vagy amíg a lánc végére nem ér.
6. Kliensnek kész kell állnia ezekre:
 - a. Egyetlen elem a láncban.
 - b. Előfordulhat, hogy egyes kérések nem érik el a lánc végét.
 - c. Kezeletlenül értek a lánc végére.

Chain of Responsibility használjuk, ha

- Több, mint egy objektum kezelhet le egy kérést és a kérést kezelő példány alaptól nem ismert, automatikusan kell megállapítani, hogy melyik objektum lesz az.
- Egy kérést objektumok egy csoportjából egy objektumnak akarjuk címezni, a fogadó konkrét megnevezése nélkül.
- Egy kérést lekezelő objektumok csoportja dinamikusan jelölhető ki.

Chain of Responsibility előnyök és hátrányok

- **Előnyök**
 - o Kérések kezelésének sorrendje szabályozható.
 - o Single Responsibility elv, Open/Closed elv
- **Hátrány**
 - o Néhány kérés kezeletlenül maradhat

Visitor (Behavioral pattern)

- Lehetővé teszi, hogy az algoritmusokat elválasszuk azokról az objektumokról, amiken azok működnek.
- **Probléma**
 - o Hívó és hívott szétválasztása.
 - o Hívó tudhat a hívottról, de fordítva tilos.
 - o A hívott dönthessen róla, hogy lehet-e vele dolgozni éppen.
- **Megoldás**
 - o Interfészeken át éri el egymást.
 - o Hívottnak legyen Accept() metódusa
 - o Hívónak legyen Visit() metódusa
 - o A hívott az Accept() metódusban döntést hoz és egyben meghívja a hívó Visit metódusát.

Visitor használjuk, ha

- Ha egy komplex objektumstruktúra (például object tree) összes elemén végre kell hajtani egy műveletet.
- Kiegészítő viselkedések üzleti logikájának (business logic) „tisztítására”.
- Ha egy behavior-nak csak az osztályhierarchia egyes osztályaiban van értelme, de más osztályokban nincs.

Visitor implementálása

1. Visitor interfész deklarálása „visiting” metódusokkal.
2. Element interfészének deklarálása.
 - a. Ha egy meglévő elemosztály-hierarchiával dolgozunk, adjuk hozzá a hierarchia alaposztályához az absztrakt Accept() metódust.
 - b. Ennek a metódusnak argumentumként egy látogató objektumot kell elfogadnia.
3. Accept() metódusok végrehajtása.
 - a. Ezeknek a metódusoknak át kell irányítaniuk a hívást a bejövő visitor objektum metódusára, ami megfelel az aktuális elem osztályának.
4. A visitor-oknak ismerniük kell a Visit() metódusok paramétertípusaiként hivatkozott összes konkrét elemosztályát.
5. Minden olyan behavior-höz, ami nem valósítható meg az elemhierarchián belül, akkor hozzon létre egy új konkrét visitor osztályt és meg kell valósítani az összes visit metódust.
6. A kliensnek visitor objektumokat kell létrehoznia és azokat az accept (elfogadó) metódusokon keresztül átadni az elemekbe.

Visitor előnyök és hátrányok

- **Előnyök**
 - o Open/Closed elv
 - o Single Responsibility elv
- **Hátrányok**
 - o Minden alkalommal frissíteni kell az összes visitor-t, amikor egy osztály hozzáadódik az elemhierarchiához vagy eltávolításra kerül belőle.
 - o Előfordulnak, hogy a visitor-ok nem rendelkeznek a szükséges hozzáféréssel azon elemek privát mezőire és metódusaihoz, amikkel dolgozniuk kell.