10. Viselkedési tervezési minták I

Iterator (Behavioral pattern)

- Szekvenciális hozzáférést biztosít 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 visszaállás
        - **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 handerre 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 használjuk, ha

- Több, mint egy objektum kezelhet le egy kérést és a kérést kezelő példány alapból 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**
  - Kérések kezelésének sorrendje szabályozható.
  - Single Responsibility elv, Open/Closed elv
- **Hátrány**
  - Néhány kérés kezeletlenül maradhat

Visitor (Behavioral pattern)

- Lehetővé teszi, hogy az algoritmusokat elválasszuk azoktól az objektumoktól, amiken azok működnek.
- **Probléma**
  - Hívó és hívott szétválasztása.
  - Hívó tudhat a hívottról, de fordítva tilos.
  - A hívott dönthessen róla, hogy lehet-e vele dolgozni éppen.
- **Megoldás**
  - Interfészeken át érjék el egymást.
  - Hívottnak legyen Accept() metódusa
  - Hívónak legyen Visit() metódusa
  - 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 előnyök és hátrányok

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

Observer (Behavioral)

- Hogyan tudják az objektumok értesíteni egymást állapotuk megváltozásáról anélkül, hogy függőség lenne a konkrét osztályaiktól.
- **Az Observer az egyik leggyakrabban használt minta!**
- **Probléma**
    - Vevő szeretne vásárolni egy új terméket, de nem szeretne mindennap meglátogatni az üzletet, ahol lehet kapni.
    - Az üzlet pedig nem szeretné feleslegesen fogyasztani az erőforrásait abból a szempontból, hogy minden egyes új termék miatt küldözget emailt, mert ez csak spam lenne.
    - Tehát a vevő pazarolja a saját idejét vagy az üzlet az erőforrásait pazarolja.
- **Megoldás**
    - Kell egy subscriber, amivel feliratkozunk valamire és az értesít.
    - Feliratkozó osztályok megvalósítanak egy ISubscriber interfészt.
    - Írjon elő egy StateChange() vagy Update() metódust.
    - A subject kezelje a feliratkozókat Subscribe(), UnSubscribe()
    - Állapotváltozáskor hívja meg az összes feliratkozó StateChange() metódusát.
    - A feliratkozók tegyék meg a frissítési lépéseket.

Observer használjuk, ha

- Egy objektum megváltoztatása maga után vonja más objektumok megváltoztatását és nem tudjuk, hogy hány objektumról van szó.
- Egy objektumnak értesítenie kell más objektumokat az értesítendő objektum szerkezetére vonatkozó feltételezések nélkül.

Observer előnyök és hátrányok

- **Előnyök**
    - Open/Closed elv
    - Az objektumok közötti kapcsolatokat futás közben is létrehozhatjuk.
- **Hátrányok**
    - A subscriber-eket véletlenszerű sorrendben értesíti.

Command (Behavioral pattern)

- Egy kérés objektumként való egységbezárása.
- Lehetővé teszi a kliens különböző kérésekkel való felparaméterezését, a kérések sorba állítását, naplózását és visszavonását.
- **Probléma**
    - Készítünk egy toolbar-t, amiben többféle gomb (button) található meg, amiknek mind különböző funkciójuk van.
    - Gombonként akkor külön alosztályokat kellene létrehozni.
    - Kód duplikáció is előfordulhat.
- **Megoldás**
    - Használjunk rétegzést, ezáltal külön választjuk a GUI-t és a business logic-ot.
    - A GUI felel a renderelésért, a business logic pedig végrehajtja a funkcionalitást.

Command használjuk, ha

- A strukturált programban callback függvényt használnánk, OO programban használjunk commandot helyette.
- Szeretnénk a kéréseket különböző időben kiszolgálni.
    - Ilyenkor várakozási sort használunk, a command-ban tároljuk a paramétereket, majd akár különböző folyamatokból/szálakból is feldolgozhatjuk őket.
- Visszavonás támogatására (eltároljuk az előző állapotot a command-ban).

Command implementálása

1. Command interfész deklarálása egy végrehajtási metódussal.
2. Interfészek implementálása az osztályokban.
    a. Minden osztálynak rendelkeznie kell a kérés paramétereinek tárolására szolgáló mezőkkel és a tényleges receiver objektumra való hivatkozással.
    b. A command konstruktorán keresztül kell inicializálni.
3. A sender osztályokhoz adjuk hozzá a parancsok tárolására szolgáló mezőket.
    a. A sender osztályok csak a command interfészen keresztül kommunikáljanak a commandjaikkal.
4. Commandok végrehajtása
5. A kliensnek ilyen sorrendben kell végrehajtania az objektumok inicializálást:
    a. Receiver-ek létrehozása
    b. Commandok létrehozása
    c. Senderek létrehozása

Command előnyök és hátrányok

- **Előnyök**
    - Elválasztja a parancsot kiadó objektumot attól, amelyik tudja, hogyan kell lekezelni.
    - Kiterjeszthetővé teszi a Command specializálásával a parancs kezelését.
    - Összetett parancsok támogatása.
    - Egy parancs több GUI elemhez is hozzárendelhető.
    - Könnyű hozzáadni új commandokat, mert ehhez egyetlen létező osztályt sem kell változtatni.
- **Hátrányok**
    - A kód bonyolultabbá válhat, mivel egy teljesen új réteget vezetünk be a sender és a receiver közé.