Gang-of-Four tervezési minták 4

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.

Mediator (Behavioral pattern) (miért szükséges és hogyan kell elkerülni a kétirányú függőségeket?)

- Olyan objektumot definiál, ami egységbe zárja, hogy objektumok egy csoportja hogyan éri el egymást.
- **Probléma**
    - Egyirányú függőség van két réteg között, ne legyen semmilyen irányú függés.
    - Egy közvetítő osztályon keresztül lehessen csak beszélgetni két osztálynak.
- **Megoldás**
    - Megoldja, hogy az egymással kommunikáló objektumoknak ne kelljen egymásra hivatkozást tárolniuk, így biztosítja az objektumok laza csatolását.

Mediator használjuk, ha

- Ha nehéz megváltoztatni néhány osztályt, mert azok szorosan kapcsolódnak egy csomó másik osztályhoz.
- Ha egy komponenst nem tudunk újrafelhasználni egy másik programban mert túlságosan függ más komponensektől.

Mediator implementálása

1. Keressük meg azokat az osztályokat, amiket függetlenebbé szeretnénk tenni.
2. Mediator interfész deklarálása.
    a. Döntő fontosságú, ha a komponens osztályokat különböző kontextusban szeretnénk újrafelhasználni.
3. Mediator osztály megvalósítása.
4. Mediator felelhet a komponensobjektumok létrehozásáért és megsemmisítéséért.
5. A komponenseknek a mediator objektumra való hivatkozást kell tárolniuk.
    a. A kapcsolat létrehozása általában a komponens konstruktorában történik, ahol a mediator objektumot adjuk át paraméterként.
6. Komponensek kódját módosítsuk úgy, hogy a többi komponens metódusai helyett a mediator értesítési metódusát hívják meg.

Mediator előnyök és hátrányok

- **Előnyök**
    - Single Responsibility elv
    - Open/Closed elv
    - Komponensek közötti kapcsolatok minimalizálása.
    - Könnyebben újrafelhasználható komponensek
- **Hátrányok**
    - Egy idő után a mediator-ből god object lehet.

Single Responsibility elv

- Minden osztály egy dologért legyen felelős és azt jól lássa el.
- **Ha nem követjük, akkor:**
    - Spagetti kód, átláthatatlanság
    - Nagy méretű objektumok
    - Mindenért felelős alkalmazások és szolgáltatások

Interpreter (Behavioral pattern)

- **Probléma**
    - Tetszőleges bemenetből tetszőleges kimenetet szeretnénk gyártani.
    - Például egy (3 + 4) – (2 + 2) stringből egy intet, aminek az értéke 3.
    - Értelmező programok írásának OOP reprezentációja az Interpreter minta.
- **Megoldás (Egyben implementálása is)**
    - Elkészítjük az írásjeleket reprezentáló osztályokat (Token)
    - Lexer elkészítése
    - Parser elkészítése

Interpreter használjuk, ha

- Ha a nyelv nyelvtana nem bonyolult.
- Ha a hatékonyság nem prioritás

Interpreter előnyök és hátrányok

- **Előnyök**
  - Könnyű megváltoztatni és bővíteni a nyelvtant.
- **Hátránya**
  - Nem hatékony

Memento (Behavioral pattern)

- Lehetővé teszi, hogy elmentse vagy visszaállítsa egy objektum előző állapotát anélkül, hogy felfedné az implementáció részeit.
- **Probléma**
  - Készítünk egy text editor alkalmazást, ahol különböző funkciókat implementálunk.
  - Biztosítani kell azt, hogy lehessen visszaállítani korábbi „állapotot/snapshotot”, ezt így menteni kell.
- **Megoldás**
  - Egységbezárás megsértése nélkül a külvilág számára elérhetővé tenni az objektum belső állapotát.
    - Így az objektum állapota később visszaállítható.

Memento implementálása

1. Originator osztály létrehozása
2. Memento osztály létrehozása, ahol hozzuk létre ugyanazokat a field-eket, amik az Originator osztályban vannak.
3. A Memento osztálynak nem szabad változtathatónak lennie (immutable), így csak konstruktoron keresztül kaphat értékeket.
4. Metódus hozzáadása, ami visszaadja a korábbi állapotot Originator osztályba, ami Memento objektumot várhat paraméterként.
5. Caretaker gondoskodik a tárolásról, ami tárolja az állapotokat, eldönti, hogy mikor kell visszaállítani.

Memento használjuk, ha

- Egy objektum (rész)állapotát később vissza kell állítani és egy közvetlen interfész az objektum állapotához használná az implementációs részleteket, vagyis megsértené az objektum egységbezárását.

Memento előnyök és hátrányok

- **Előnyök**
  - Megőrzi az egységbezárás határait.
- **Hátrányok**
  - Erőforrásigényes
  - Nem mindig jósolható meg a Caretaker által lefoglalt hely

State (Behavioral pattern)

- Lehetővé teszi egy objektum viselkedésének megváltozását, amikor megváltozik az állapota.
- **Probléma**
  - Túl nagy switch-case szerkezet, sok állapot = sok ellenőrzés
- **Megoldás**
  - Kontextust hozunk létre, ami az egyik állapotra hivatkozást tárol.

State implementálása

1. Hozzunk létre egy osztályt, ami lesz a kontextus (context).
2. State interfész létrehozása, hozzuk létre az állapot-specifikus viselkedést tartalmazó metódusokat.
3. Minden aktuális állapothoz hozzunk létre egy osztályt, ami implementálja a State interfészt.
4. Context osztályban deklaráljunk egy referencia mezőt a State interfész típusához, aminek legyen egy publikus setter-je, amivel felül lehet írni az értékét.
5. Megfelelő állapotfeltételhez hívjuk meg a megfelelő metódust.
6. Kontextus állapotának megváltoztatásához létre kell hozni egy példányt az egyik state osztályból és azt adjuk át a kontextusnak.

State használjuk, ha

- Az objektum viselkedése függ az állapotától és a viselkedését az aktuális állapotnak megfelelően futás közben meg kell változtatnia.
- A műveleteknek nagy feltételes ágai vannak, amik az objektum állapotától függenek.

State előnyök és hátrányok

- **Előnyök**
  - Egységbezárja az állapotfüggő viselkedést, így könnyű az új állapotok bevezetése.
  - Áttekinthetőbb kód, nincs nagy switch-case szerkezet
  - A State objektumot meg lehet osztani.
- **Hátrányok**
  - Nő az osztályok száma, így csak indokolt esetben használjuk.

Composite (Structural pattern)

- Másnéven **Object Tree**
- **Probléma**
  - Nehezen tudunk az objektumainkból hierarchikus rendszert építeni.
  - Például részlegek és dolgozók korrekt ábrázolása.
  - Egy részfa vagy akár egy levélelem is ugyanazt a szolgáltatáskészletet nyújtsa.
- **Megoldás**
  - Fa szerkezet építése
  - Egy csomópontnak tetszőleges mennyiségű gyermekeleme legyen.
  - A csomópontnak és levél elemek is ugyanazt az interfészt valósítsák meg.
  - Lehessen rekurzívan bejárni.

Composite implementálása

1. Alkalmazás alapvető modellje fa struktúraként ábrázolható kell legyen.
2. Komponens interfész implementálása
3. Levélosztály létrehozása az egyszerű elemek ábrázolására.
4. Osztály létrehozása az összetett elemek ábrázolásához.
    a. Tömböt létre kell hozni, amiben az alelemekre való hivatkozásokat tárolja.
    b. Tömbnek képesnek kell lennie a levelek, konténerek tárolására is, ezért a komponens interfész típusával kell deklarálni.
5. Metódusok deklarálása, amivel hozzáadhatunk vagy törölhetünk gyermekelemeket.

Composite használjuk, ha

- Objektumok rész-egész viszonyát szeretnénk kezelni.
- A kliensek számára el akarjuk rejteni, hogy egy objektum egyedi objektum vagy kompozit objektum.
    - Bizonyos szempontból egységesen szeretnénk kezelni őket.

Composite előnyök és hátrányok

- **Előnyök**
    - Összetetteb fa struktúrával is dolgozhatunk.
    - Open/Closed elv
- **Hátrányok**
    - Nehéz lehet közös interfészt biztosítani, mivel a funkcionalitások eltérhetnek.