11. Viselkedési tervezési minták II

# Strategy (Behavioral pattern)

* Algoritmusok egy csoportján belül az egyes algoritmusok egységbezárása és egymással kicserélhetővé tétele.
* A kliens szemszögéből az általa használt algoritmusok szabadon kicserélhetőek.
* **Probléma**
  + Készítünk egy navigációs alkalmazást, ahol időről időre új funkciókat szeretnénk lefejleszteni.
  + Útválasztási algoritmusokat akarunk fejleszteni autókhoz, sétához, stb.
  + Átláthatatlan lett a kód, mert amikor hozzáadunk mindig egy új útválasztási algoritmust, megduplázódott a kód.
* **Megoldás**
  + Adott algoritmust szervezzük ki egy külön osztályba, ezt nevezzük **stratégiának.**

## Strategy használjuk, ha

* Egy objektumon belül egy algoritmus különböző változatait szeretnénk használni és képesnek kell lennie a változásra az egyik algoritmusról a másikra futás (runtime) közben.
* Sok hasonló osztály van, amik csak abban különböznek egymástól, hogy hogyan hajtanak végre bizonyos viselkedést.
* Business Logic elkülönítése

## Strategy implementálása

1. Context osztályban azonosítsunk be egy olyan algoritmust, ami hajlamos a gyakori változásokra.
2. Strategy interfész deklarálása
3. Egyenként implementáljuk a Strategy interfészt a megfelelő algoritmussal a saját osztályába.
4. Context osztályban legyen egy field, ami tárolja a strategy objektum referenciáját tárolja.
   1. Setter-t is állítsunk neki

## Strategy előnyök és hátrányok

* **Előnyök**
  + Objektumon belül algoritmust tudunk cserélni futásidőben. (runtime)
  + Az algoritmus implementálását izolálhatjuk az azt használó kódtól.
  + Öröklődést kicserélhetjük kompozícióval.
  + Open/Closed elv
* **Hátrányok**
  + Ha csak pár algoritmus van és azok is ritkán változnak, akkor nem érdemes túlbonyolítani új algoritmusokkal, osztályokkal és interfészekkel, amik a mintával együtt járnak.

# Template (Behavioral)

* Egy műveleten belül algoritmus vázat definiál és ennek néhány lépésének implementálását a leszármazott osztályra bízza.
* **Probléma**
  + Készítünk egy olyan alkalmazást, amivel különböző dokumentumokból adatokat lehet kinyerni.
  + Egy idővel rájövünk, hogy például a PDF, CSV fájlok között viszonylag hasonló műveletek hajtódnak végre, így kód duplikáció keletkezhet.
* **Megoldás**
  + Magát az algoritmust bontsuk szét kisebb lépésekre, metódusokra.
  + Ezeket fogjuk meghívni a template method-ban.

## Template használjuk, ha

* Algoritmust kisebb lépésekre szeretnénk bontani.
* Logikai hasonlóság esetén

## Template implementálása

1. Kisebb részekre bontás
2. Absztrakt osztály létrehozása, ahol deklaráljuk a template method-ot.
3. Hívjuk meg az alosztályokat, a lépéseket a template method-ban.

## Template előnyök és hátrányok

* **Előnyök**
  + Kód duplikáció elkerülhető vele, tehát a hierarchiában a közös kódrészeket a szülő osztályban egy helyen adjuk meg (template method), ami a különböző viselkedést megvalósító egyéb műveleteket hívja meg.
    - Leszármazott osztályban felül lehet definiálni.
* **Hátrányok**
  + Megsérthetjük a Liskov behelyettesítési elvet, ha egy alosztályon keresztül elnyomja az alapértelmezett lépés implementációját.
  + Template method-okat egy idő után nehéz karbantartani, ha sok kisebb lépést (metódusokat) tartalmaz.

# 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.

# 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