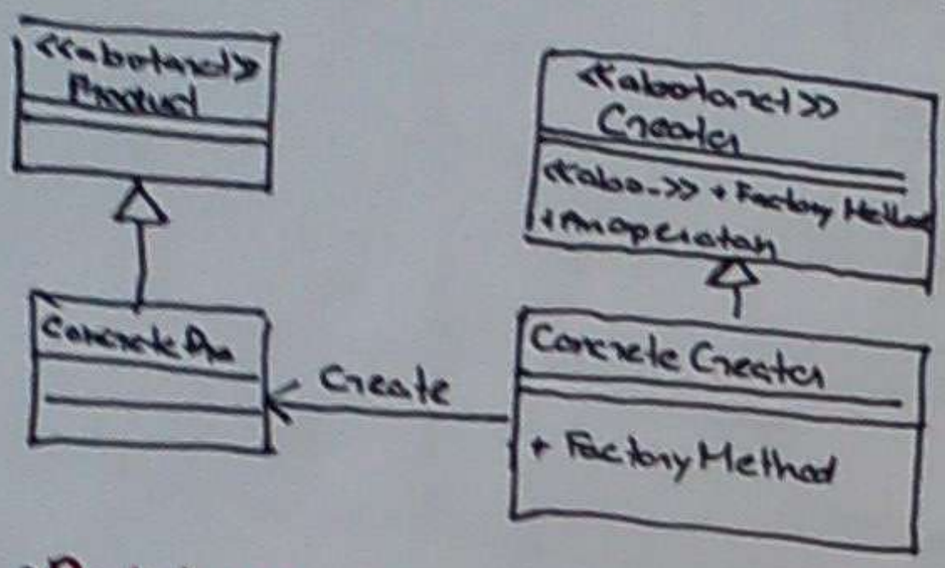
Design Patterns

• Létrehozási

- **Factory Method**: lehetővé teszi, hogy az új példány létrehozását a leszármazott osztályra bízzuk. „Virtuális Konstruktor"

pl.: Framework, ami egyszerre több objektum kezelését támogatja



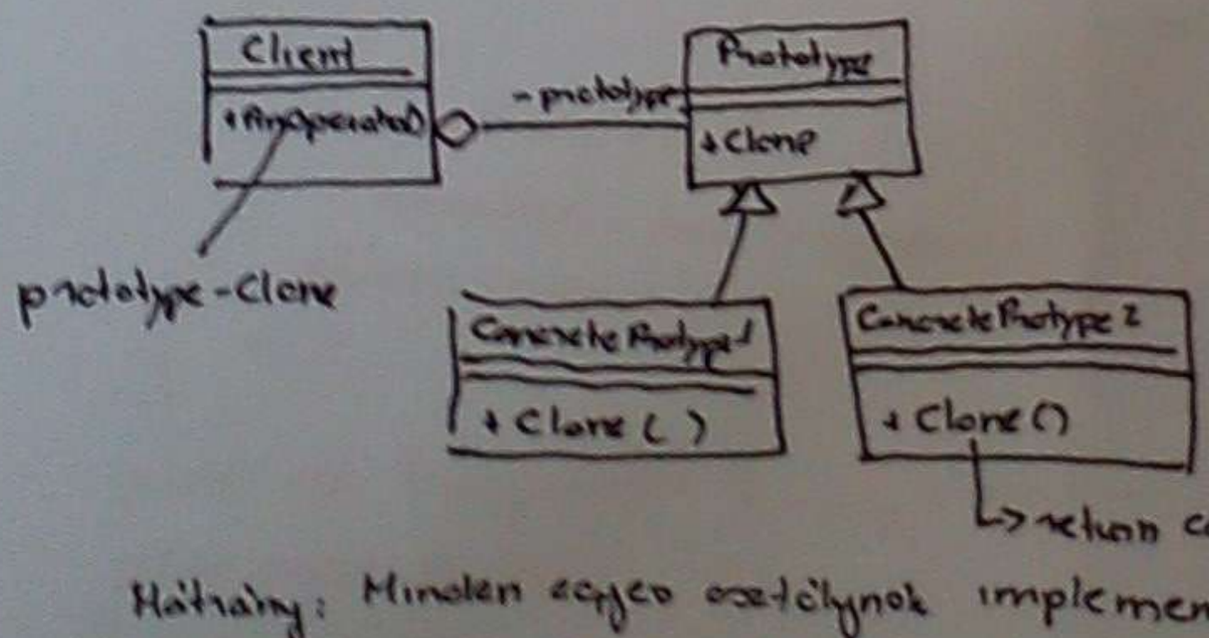
* Ha egy osztály nem látja előre annak az objektumnak az osztályát amit létre kell hoznia.

* Ha egy osztály azt szeretné, hogy leszármazottai határozzák meg azt az objektumot, amit létre kell hoznia.

- **Prototype**: prototípus alapján új objektumpéldányok készítése
  - az obj. nagyon összetett és egyszerűbb másolni
  - rugalmasan kívánjuk létrehozni

* Ha egy rendszernek függetlennek kell lennie a létrehozandó objektumok típusaitól és

  - ha a példányosítandó osztályok futási időben határozhatók meg
  - ha nem akarunk nagy párhuzamos hierarchiákat
  - ha könnyebb másolni:



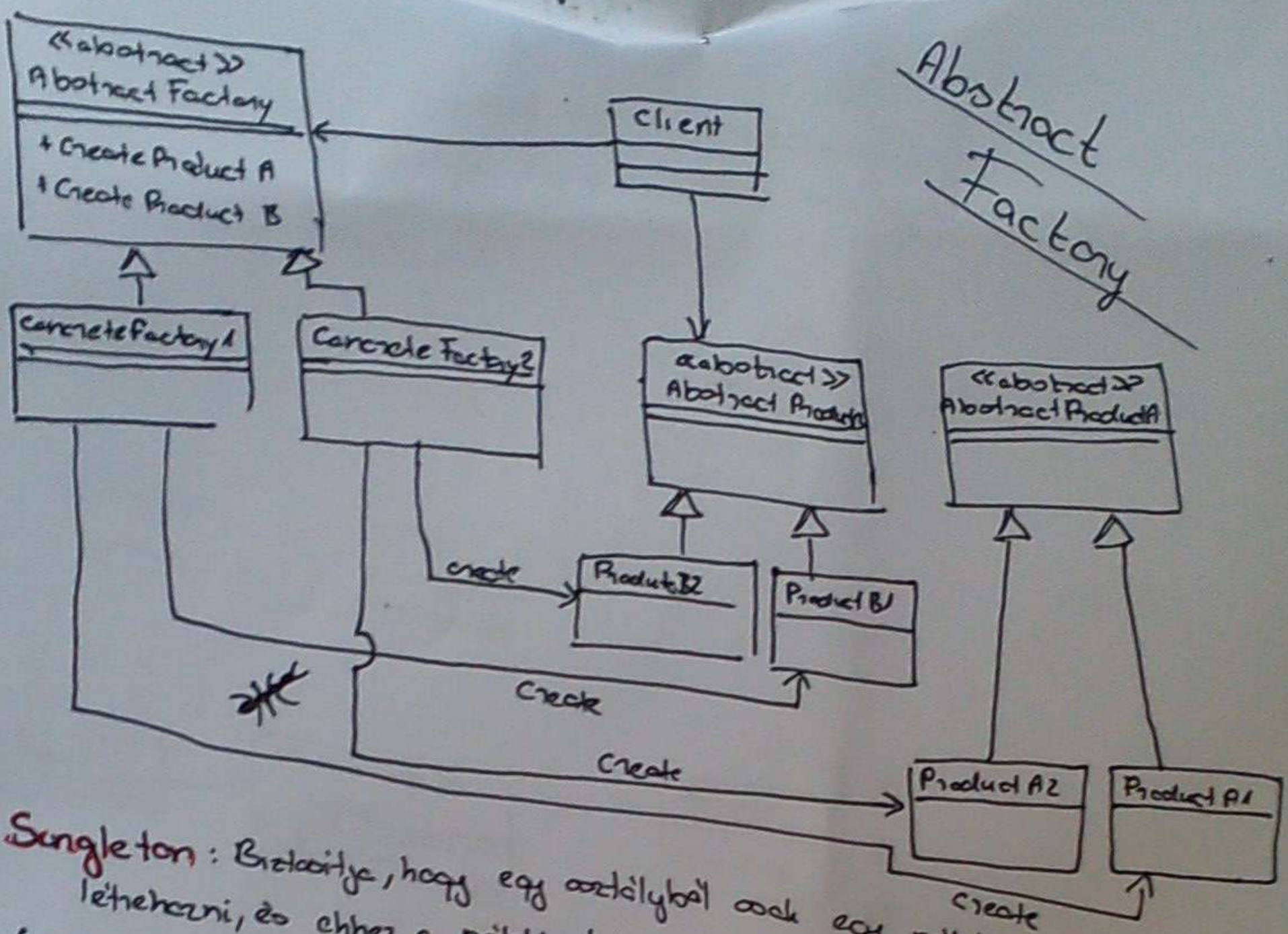
Előny: • objektumok hozzáadása és elvétele futási időben
• új, változó struktúrájú objektumok létrehozása
• Redukált számosságú, kevesebb alosztály

Hátrány: Minden egyes osztálynak implementálnia kell a Clone() dmyf.

- **Absztract Factory**: rendszernek függetlennek kell lennie az általa létrehozott dolgoktól, több termékcsaládból kell együtt működnie,

Előny: • Elrejtjük a konkrét osztályokat
• a termékcsaládokat könnyű kicserélni
• Előírja a termékek közötti konzisztenciát

Hátrány: nehéz új termék hozzáadása, mert az egész hierarchiát módosítani kell



- **Singleton**: Biztosítja, hogy egy osztályból csak egy példányt lehessen létrehozni, és ehhez a példányhoz globális hozzáférést biztosít
(Ablakkezelő obj., Fájlrendszer obj.)

```
public class Singleton
{
    private static Singleton instance = null;
    public static Singleton Instance    // csak a Statikus Instance
    {                                   metóduson keresztül lehet példányt
        get                             létrehozni
        {
            if (instance = null)
                instance = new Singleton();
            return instance;
        }
    }

    protected Singleton() {...}; // konstruktor protected
    public void Print() {...};
}

Singleton s1 = Singleton.Instance;
...
Singleton.Instance.Print();
```

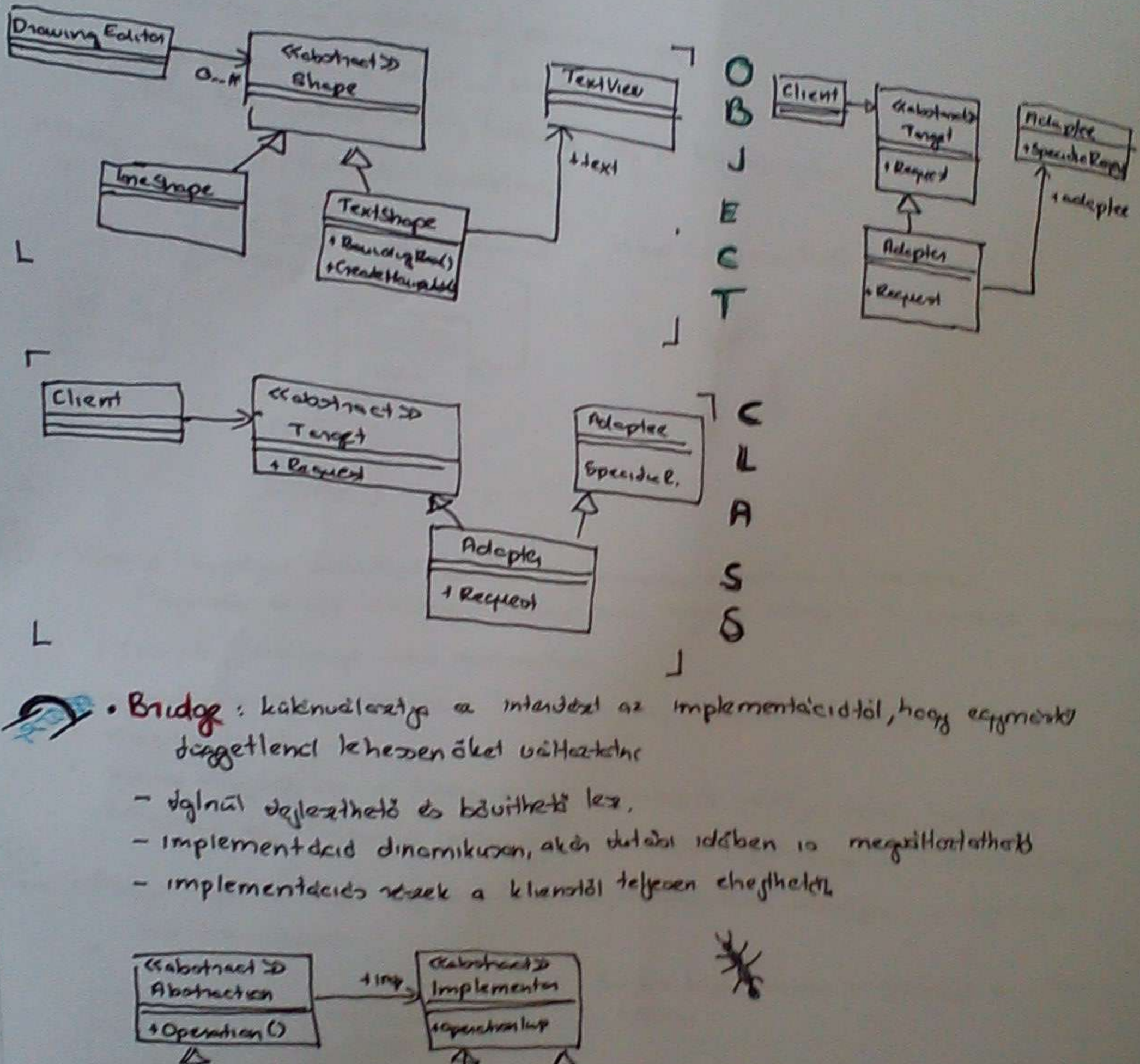
A létrehozási minták célja: rendszer rugalmasabb, kevésbé bonyolult, a meglévő osztályok ~~megváltozása nélkül~~ könnyebben bővíthető.

Modern nyelvekben reflexiós technikák segítségével kiváltható.

Strukturális:

- **Adapter**: egy osztály interfészét olyan interfészé konvertálja, amit a kliens vár. Lehetővé teszi olyan osztályok együttműködését amik egyébként inkompatibilisek

class és object adapter

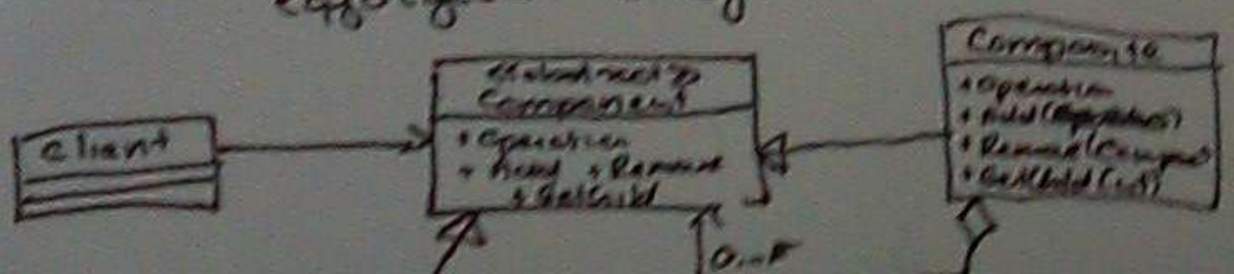


- **Bridge**: különválasztja az interfészt az implementációtól, hogy egymástól függetlenül lehessenek őket változtatni

1)

- ügynül cserélhető és bővíthető lesz.
- implementáció dinamikusan, akár futási időben is megváltoztatható
- implementációs részek a klienstől teljesen elrejthetők

- **Composite**: rész-egész viszonyban álló objektumokat datastuktúrába rendez. Kliensek számára elérhetővé tesz, hogy egyszerű és komplex objektumokat egységesen kezelje



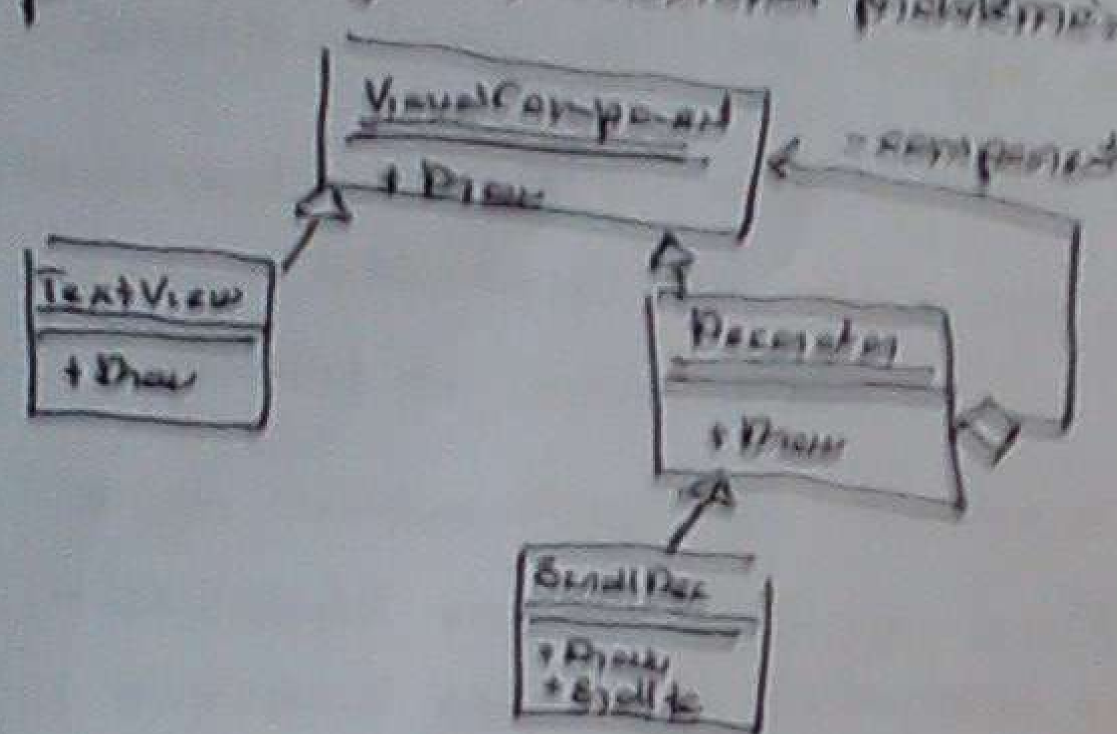
16. • **Decorator** (Toldalék): objektumok funkciójának dinamikus kiegészítése
Rugalmas alternatívája a leszármaztatásnak.

× dinamikusan szeretnénk funkcionalitást hozzárendelni az egyes objektumokhoz de a kliens számára átlátszó módon

× származtatás nem praktikus, de decorator igen.

Előny: • rugalmasabb mint a statikus öröklés
• több testreszabható osztály helyett meg a hierarchiát

Hátrány: • bonyolult, reflexiónál problémás



VisualComponentből származik

• **Facade**: egységes interfészt definiál egy alrendszer interfészeinek halmazához.
Magasabb szintű interfészt definiál, ami könnyebbé teszi az alrendszer használatát

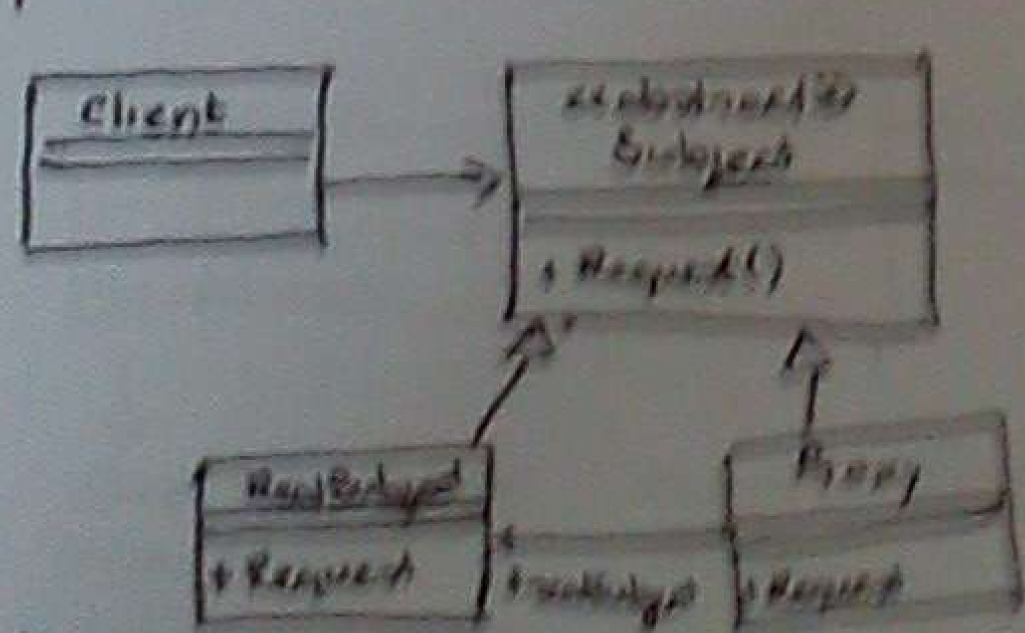
p.l: compiler, többrétegű üzleti alkalmazások

× egyszerű interfészt szeretnénk biztosítani egy komplex rendszerhez

× rétegelés esetén

× számos függőség van a kliens és az alrendszerek osztályai között. Ilyenkor ha létrehozunk egy Facade-ot elrejtjük az alrendszer függőségeit és a komplexitását.

• **Proxy**: objektum helyett egy helyettesítő objektumot, ami szabályozza az objektumhoz való hozzáférést



Subject közös interfész a RealSubject és a Proxy számára

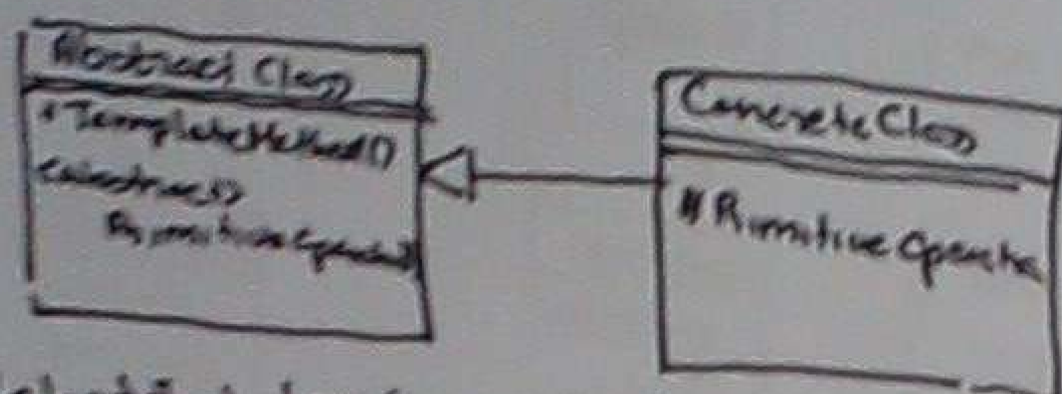
Távoli Proxy (Remote proxy): egy különböző címtérbeli objektumot lokálisan reprezentál

Védelmi proxy: hozzáférés szabályozása

Smart Pointer: egy pointer helyettesítése, hogy automatikus műveleteket hajtson végre

Viselkedési minták:

• **Template method**: egy műveleten belül algoritmusvázat definiál és ennek néhány lépésének implementálását a leszármazott osztályra bízza.



Lehetővé teszi, hogy az algoritmus invariáns részeit egyhelyen definiáljuk, és a változó részeket leszármazott osztályban adjuk meg

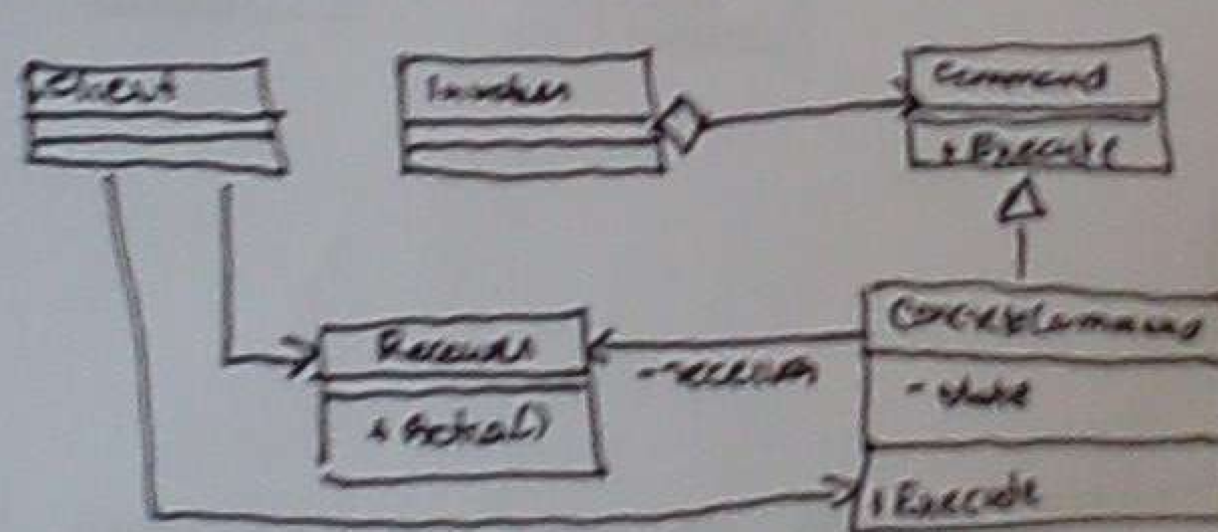
Lehetővé teszi a hook [illegible] definiálását: ezek kiterjesztési pontok a kódban.

• **Command**: egy kérés objektumként való egységbezárása. Lehetővé teszi a kliens különböző kérésekkel való felparaméterezését, a kérések sorba állítását, naplózását és visszavonását.

Nagyon rendszerfüggő a koncepció és az implementáció is.

× Strukturált program callback <-> OO command

× Különböző kéréseket különböző időben kiszolgálni.

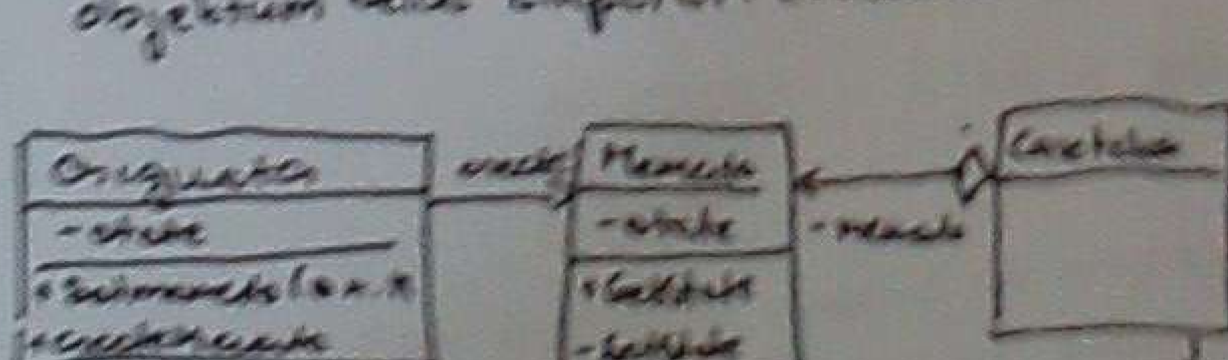


× Elválasztja a parancsot kiadó objektumot attól, amelyik tudja [illegible]

× Összetett parancsok támogatása

× Command Processor
- beépítve támogatja az undo alapjait
- nyilvántartja, controlleréi a commandokat és aktiválja

• **Memento**: egységbezárás megsértése nélkül külsőleg mentse el/elérhetővé tenni az objektum belső állapotát. (Visszaállítás)



↓ állapotát kell tudni visszaállítani

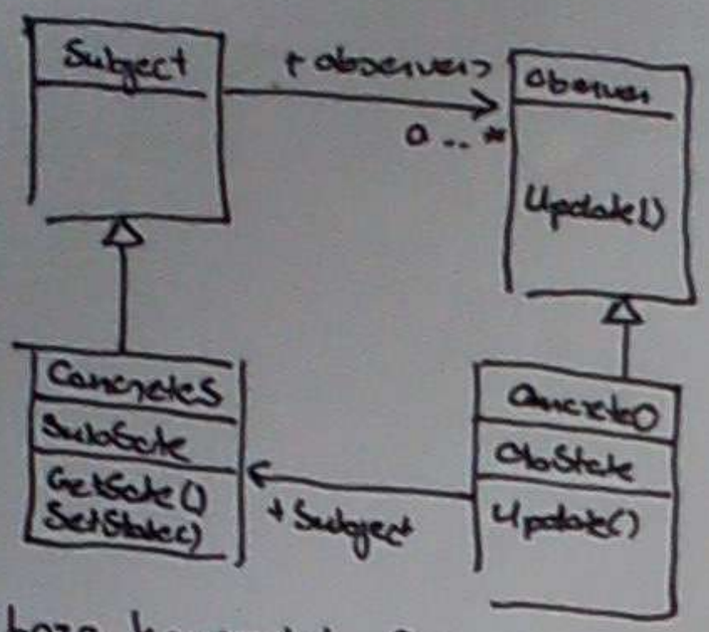
↓ originator állapotát tárolja

↓ nyilvántartja a Mementókat

Hátrány: - Sokszor erőforrás-igényes
- Caretaker által foglalt hely nem mindig jósolható meg

ODOL

• **Observer**: hogyan tudjak értesíteni egymást függőség nélkül
MVC alapja; ha értékkel um beregisztrálós

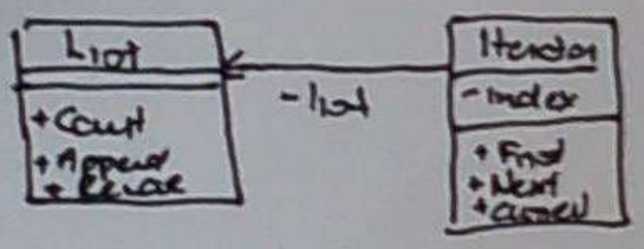


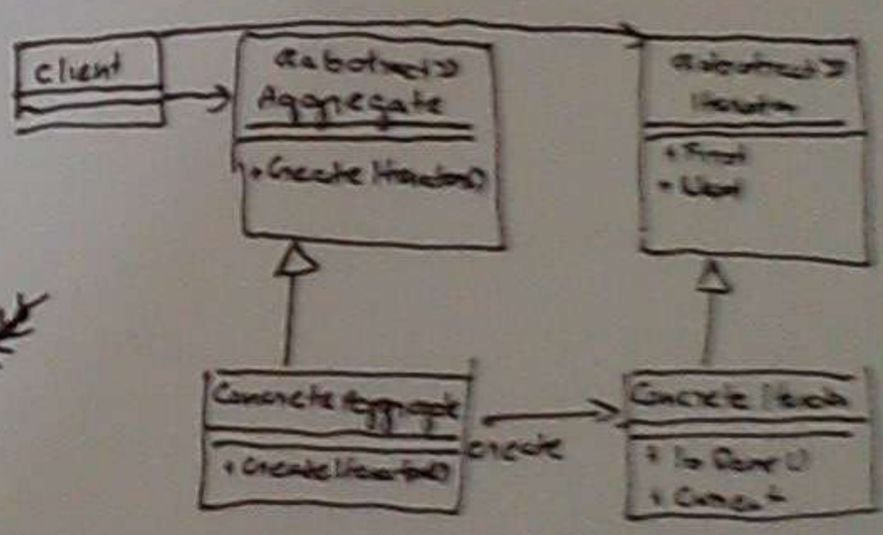
} Sub: tárolja a beregisztrált Observereket
és erre interfészt ad.
Obs: Interfészt ad azon objektumok számára
akik be akarnak regisztrálni

} Referencia a ConcreteSub-ra
Implementálja az Obs. interfészt amit a Sub
meghív ha a ConcreteSub állapot változik

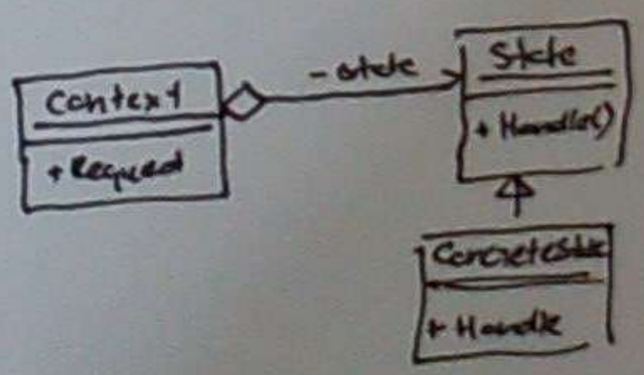
✓ Laza kapcsolat Sub és Obs között ✗ Felelőtlen Update
✓ Üzenetszórás támogatása

• **Iterator**: Szekvenciális hozzáférést biztosít egy összetett objektum elemeihez anélkül,
hogy annak belső reprezentációját feldedné



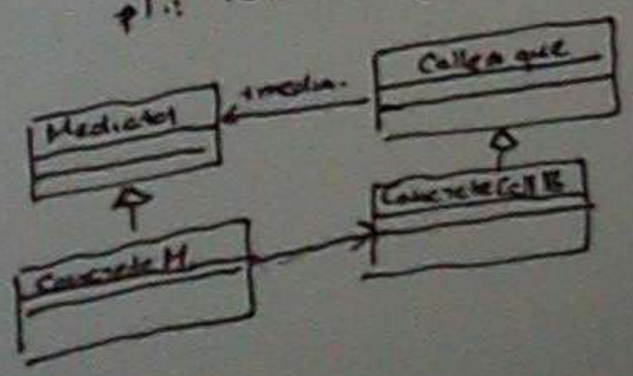


• **State**: Lehetővé teszi egy objektum viselkedésének megváltoztatását, ha annak a
állapota



✓ Egyszerűsíthetjük az állapotfüggő viselkedést, így
könnyű új állapotokat bevezetni
✓ Áttekinthetőbb kód
✗ Nő az osztályok száma

• **Mediator**: olyan objektumot definiál ami egyesítve egységbe zárja, hogy az objektumok ne
csoportja, hogyan épüljenek el egymást. (Egymással kommunikálás használatának
nélkül ⇝ laza csatolás.)
pl.: Form vagy dialógus ablak



- **Strategy**: algoritmusok egy csoportjon belül az egyes algoritmusok egységbe zárása
  és egymással kicserélhetővé tétele.