Szoftvertechnikák

Architekturális tervezés
Tervezési minták
Bináris komponensek, reflexió

Elméleti összefoglaló a 9-13. előadások tartalma alapján[1]

Petrovics Kornél

2017. május 23.

[1] Forrás: Benedek Zoltán, Dr Levendovszky Tihamér: Szoftverarchitektúrák c. előadás prezentáció
Benedek Zoltán: Tervezési minták (design patterns) c. előadás prezentáció
Benedek Zoltán: Bináris komponensek, reflexió c. előadás prezentáció

Architekturális tervezés (ea. 9-10):

Az architektúra a szoftverrendszer szervezése, strukturálása.

Az architektúrát a modelleken keresztül ismerhetjük meg, azt azok specifikálják. (Unified Process az architektúra folyamatos finomítása a modellek segítségével – usecase, analízis, tervezés, telepítés, implementáció)

Az architektúra és a funkcionalitás a megfelelő egyensúlyban legyen.

Bevált, ismert architektúrák: **Rétegek, Document-view architektúra, MVC architektúra, csővezetékek és szűrők**

Rétegek:

lásd pl. tcp/ip protokol

Az i. réteg: szolgáltatásokat nyújt az i+1. rétegnek. A saját szolgáltatásait az i-1. réteg szolgáltatásaira építve valósítja meg.

layer – logikai réteg
tier – fizikai réteg

előnyei:

- Egymástól független komponensek rétegekbe csoportosítása → a részek egymástól függetlenül cserélhetők, fejleszthetők
- túl komplex a feladat → vezessünk be egy új absztrakciós szintet (réteget)
- rétegek újrafelhasználhatósága (egy rétegre több szolgáltatás is építhető)

hátrányok:

- egy egyszerű feladatnál felesleges komplexitás
- → teljesítményromlás

Információs rendszerek (vállalati rendszerek) rétegei:

- Két rétegű architektúra
  - kliens-szerver / alkalmazások-megosztott adatbázis
  - A kliens alkalmazások közvetlen hozzáférnek az adatbázishoz
  - **az adat megosztott, a megjelenítés elosztott**

  előnyök:

  - a már meglévő adatokat több szempontból is megjeleníthetjük
  - adat, alkalmazás elkülönül

  hátrány: **hova tegyük az üzleti logikát???**

  - alkalmazásba? keveredik a GUIval, több alkalmazás esetén duplikálódik...
  - adatbázisba? adatkezelő nem támogatja, ha mégis, biztos nem objektumorientáltan

- Három rétegű architektúra:
  - alkalmazás-tartomány-adatbázis
  - az alkalmazás: az alkalmazás logika és a UI ide kerül
    tartomány: ez az üzleti logika
    adatbázis: belső sémával

előnyei a kétrétegű architektúrával szemben:

- az üzleti logika nincs duplikálva
- az üzleti logika könnyebben újrafelhasználható (független a klienstől) – pl. webes, vastag kliens
- az adatbázis átszervezhető az alkalmazástól függetlenül

hátránya: nő a komplexitás, az erőforrásigény

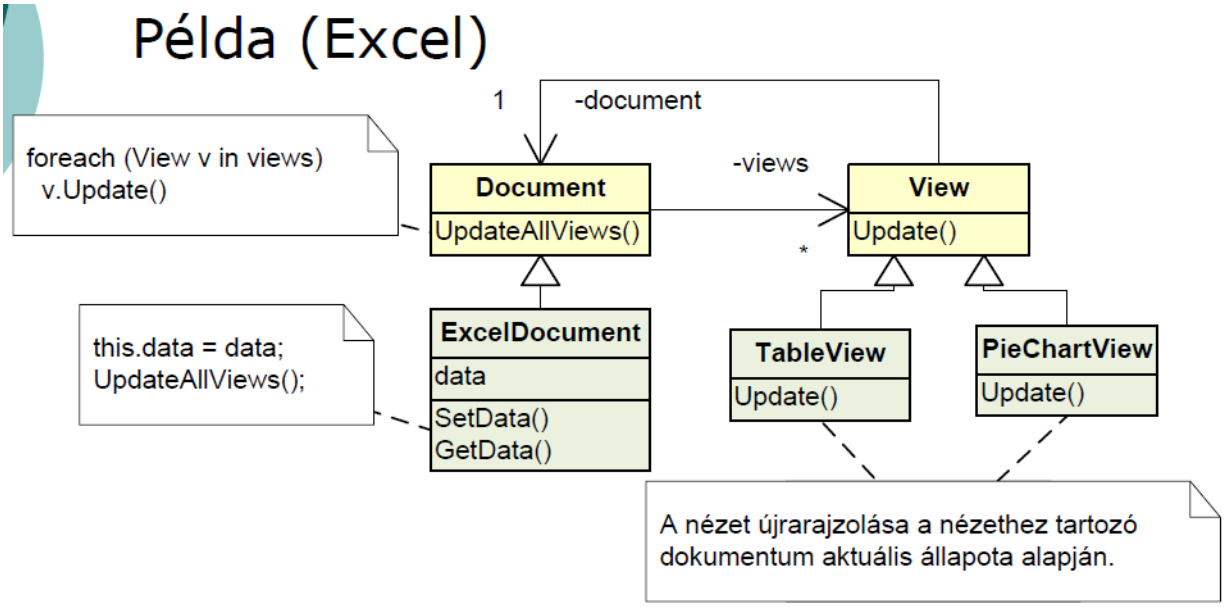
Document-view architektúra:

Válasszuk külön az adatok **kezeléséért** felelős kódot az adatok **megjelenítéséért** felelős kódtól.

Document: feladata az **adatok tárolása, menedzselése**
View: fealdata az **adatok megjelenítése, felhasználói interakciók kezelése**

Ezáltal:

- több dokumentum egyidejű megnyitása (pl. firefox tabok)
- egy dokumentumhoz több nézet rendelése – **ha valamelyik view megváltoztatja a dokumentum adatait, akkor a dokumentum értesíti az összes beregisztrált view-t a változásról**, ennek hatására azok frissítik magukat



- a modell kódjában egyetlen *View* lista, így a modell független a *View*-t implementáló osztályoktól
- egyszerű mechanizmus az egyes nézetek konzisztens állapotban tartására

MVC architektúra:

Model (Document)
View (View)
Controller (View)

A **model** tartalmazza az adatokat, valamint az azokon végrehajtható műveleteket (alkalmazáslogika). A modell egy közös **Observer** típusú listában tárolja a viewkat és contollereket, ha egy controller megváltoztatja a modellt, akkor a modell értesíti az ebben a listában tárolt V és C elemeket, hogy azok is eszerint frissítsék magukat.

A **view** megjeleníti a modellből kiolvasott adatokat.

A **Controller** kezeli a felhasználói interakciókat, itt vannak az egér/billentyűzet eseménykezelői, melyek tipikusan a modellbe hívnak bele.

Előnyei:

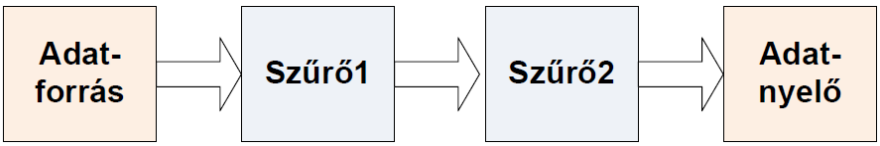
- egy adatnak (dokumentumnak) több (szinkronizált) nézete lehet
- függetlenül cserélhető, fejleszthető view-k és controller-ek

Hátrányai:

- komplexitás növekszik,
- feleslegesen sok frissítés lehet – teljesítmény csökken
- a view és controller gyakran nem különválasztható és nem is célszerű. **Megoldás: Document-view architektúra!**

Csővezetékek és szűrők: architektúrális minta adatfolyamot feldolgozó rendszerekre

Adatfolyam feldolgozásának lépéseit **szűrők** végzik el, az egyes szűrőket **csővezetékek** kötik össze. (pipe, filter) A szűrők tetszőlegesen kombinálhatók.



pl.: videófolyam feldolgozása (kontrasztállítás, világosságállítás, … formátumváltás, tömörítés)

A szűrők:

- transzformációt hajtanak végre az adaton
- aktív szűrő: a bemeneti csővezetékről behúzza az adatot, a kimeneti csővezetékre kitolja
- passzív szűrő: a bemenetére a csővezeték tolja be az adatot, a kimenetén a csővezeték húzza ki

A csővezetékek: átviszik az adatot, aktív szűrők esetén szinkronizálhatnak is.

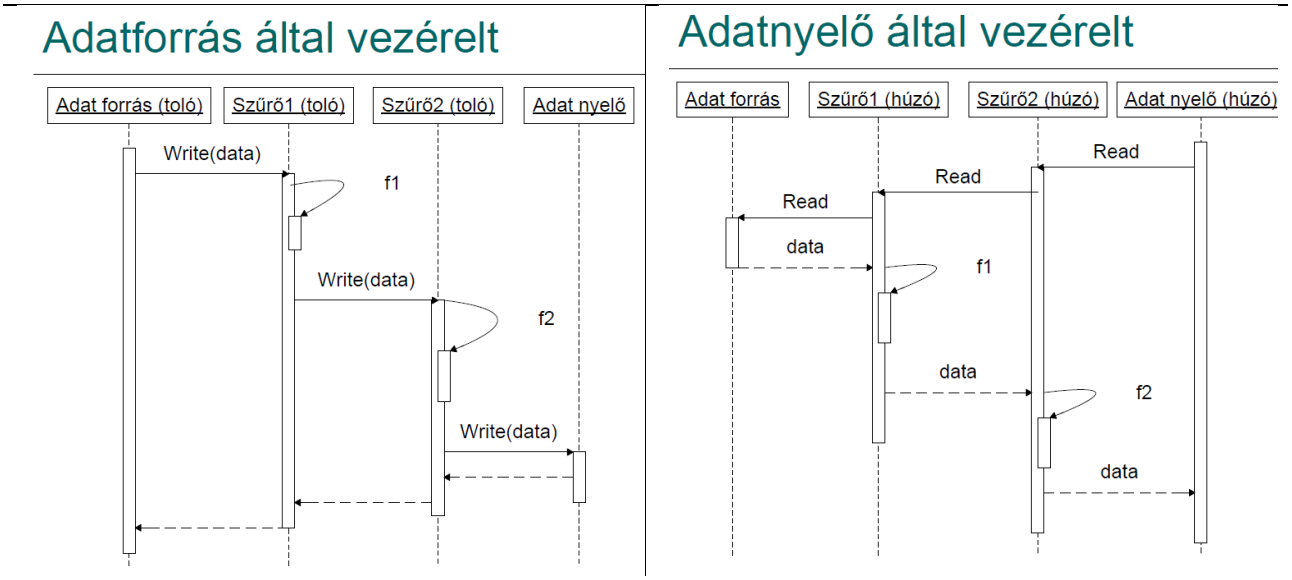
Előnyök:

- a filterek tetszőlegesen kombinálhatók, lecserélhetők, újrafelhasználhatók
- párhuzamos feldolgozás lehetősége

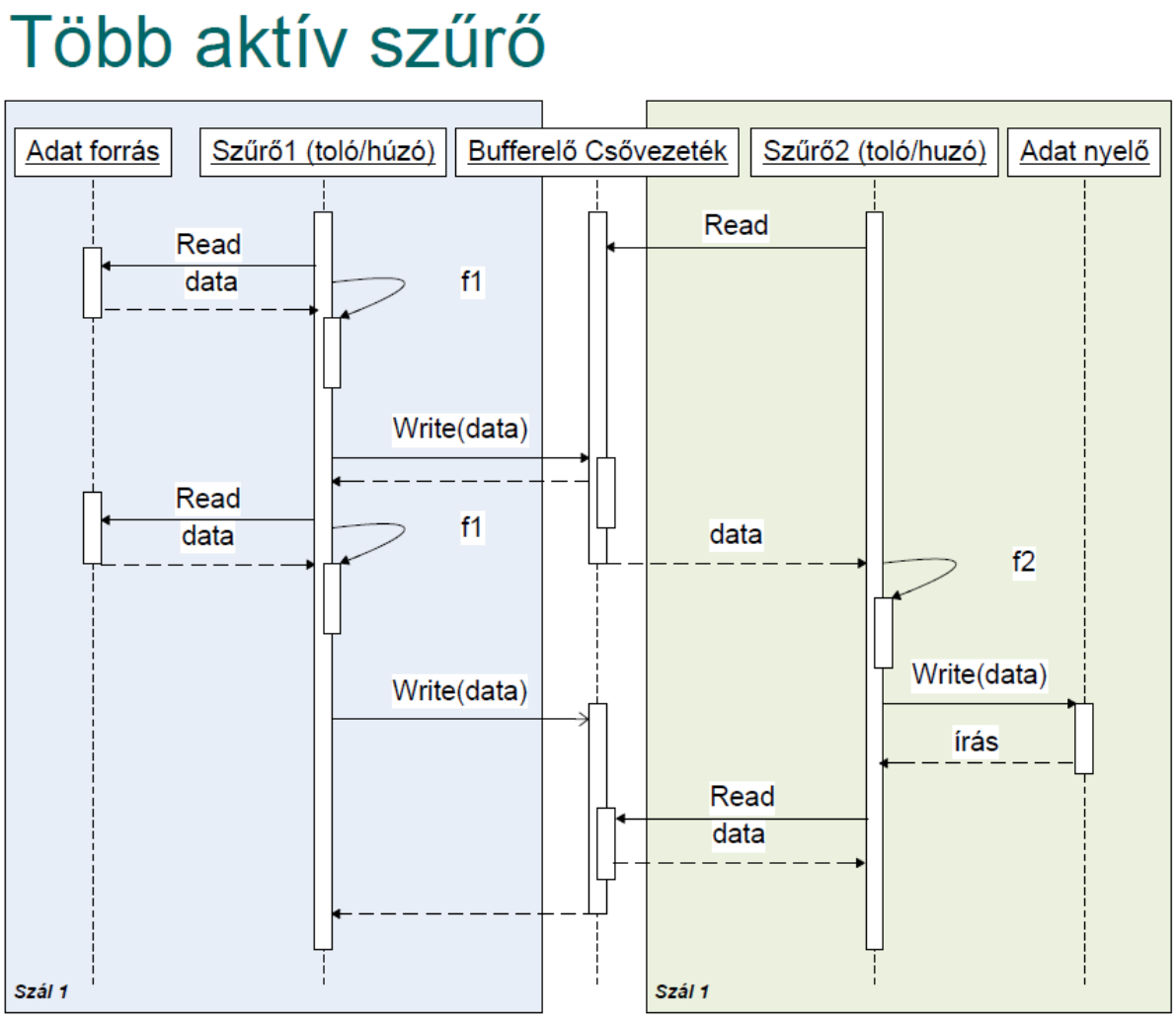
Hátrányok:

- hibakezelés nehézsége
- adattranszformációs overhead

Passzív szűrők működése:



Aktív szűrőkre példa:



Tervezési minták (ea. 11-12):

A tervezési minta (design pattern) leír egy gyakran előforduló problémát és a hozzá tartozó megoldás magját, amit alkalmazva sok esetben hatékony megoldást kaphatunk.

Tervezési minta leírása: pattern name / problem / solution / consequences

A minták csoportosítása hatókör szerint:

I.) Architektúrális minták
- ☞ előző előadáson tanultak (rétegek, MVC, Doc-View, pipe-ok…)
- ☞ alapstruktúrát adnak a rendszernek

II.) Tervezési minták
- ☞ alrendszereken, szoftverkomponenseken belül alkalmazható,
- ☞ azok kommunikációjának megoldására általános tervezési útmutatást adó minta

III.) Idiómák
- ☞ programozási nyelv specifikus minták

Miben segítenek a tervezési minták? **A tervezés fázisában**:
- ☞ megfelelő objektumok megtalálása
- ☞ újrafelhasználható kód tervezése
- ☞ könnyen fejleszthető kód tervezése

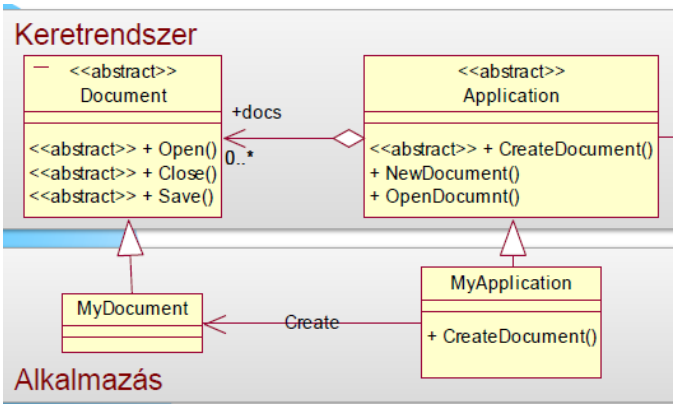
→ cél: függőségek csökkentése – egy változtatás minél kevesebb komponenst érintsen.

A tervezés során több mintát is felhasználunk (általában egy architektúrális, több tervezési) = **system of patterns.**

Létrehozási tervezési minták:

➢ Factory method:

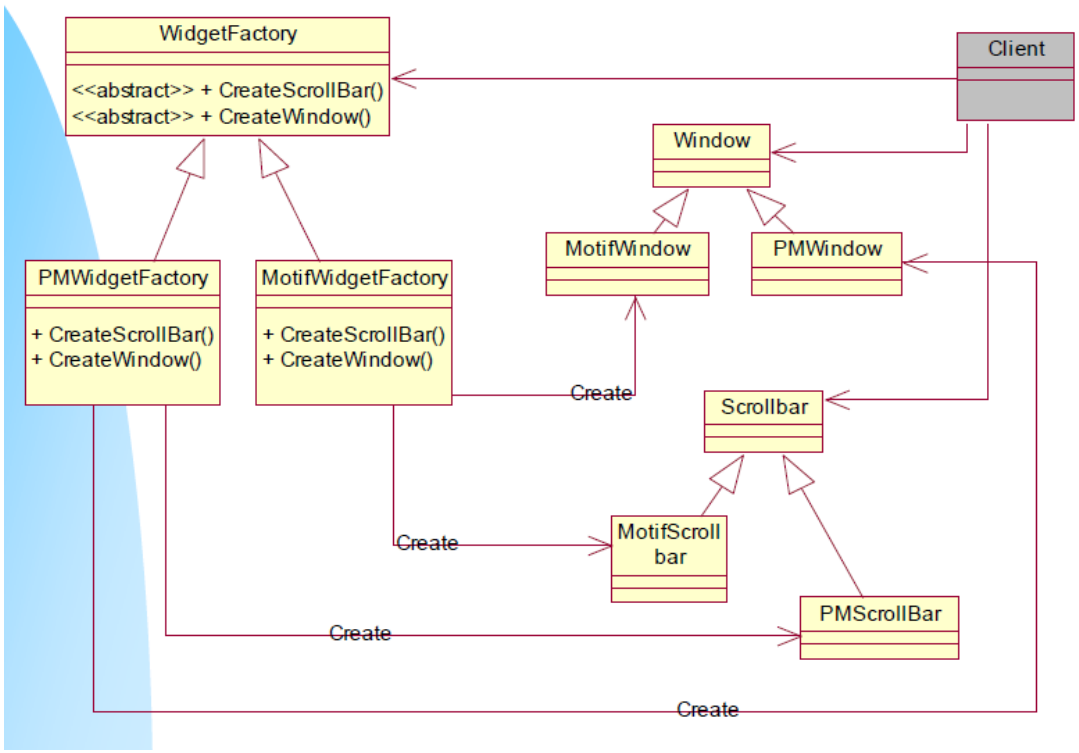
- ☞ **probléma**: definiált egy keretrendszer, pl.: absztrakt Application és Document osztályokkal. Az Application NewDoucment() függvénye a keretrendszer tervezésekor nem tudja, hogy később (felhasználástól függően) milyen dokumentumot kell létrehoznia.
- ☞ **megoldás**: felveszünk egy absztrakt CreateDocument() metódust az Applicationbe, amit a keretrendszerre építő fejlesztőnek a MyApplication osztályban úgy kell megvalósítania, hogy az MyDocument-et hozzon létre. Ezt a CreateDocument() függvényt hívja meg a NewDocument() az Application osztályban.



- ☞ használjuk ha: egy osztály azt szeretné, hogy a leszármazottai határozzák meg azt az objektumot, amit létre kell hoznia

➢ Abstract factory:

- ☞ **probléma**: az egyes GUI elemeknek környezettől függően más megjelenést szeretnénk (pl.: ScrollBar és Window megjelenése PM illetve Motif környezetben)
- ☞ **megoldás**: minden GUI vezérlőelemnek külön verziót készítünk minden megjelenési környezethez a következő módon: pl.:
  - létrehozunk egy absztrakt WidgetFactory osztályt, aminek interfésze támogatja a vizuális elemek létrehozását,
  - ennek metódusai absztrakt osztállyal térnek vissza (tehát Window és nem MotifWindow)
  - a konkrét megjelenítések implementálásához leszármazunk a WidgetFactroy osztályból, ahol az interfészt megvalósítjuk, és itt már a MotifWindow-al térünk vissza
  - a kliens egy konkrét WidgetFactory objektum (amivel fel volt konfigurálva előzetesen) megfelelő metódusával hozza létre a konkrét felületelemeket



- ☞ használjuk ha: a rendszernek több termékcsaláddal kell együttműködnie

➢ Singleton:

- ☞ **probléma:** biztosítani akarjuk, hogy egy osztályból csak egy példány lehessen, és ehhez az osztályhoz globális hozzáférésünk legyen

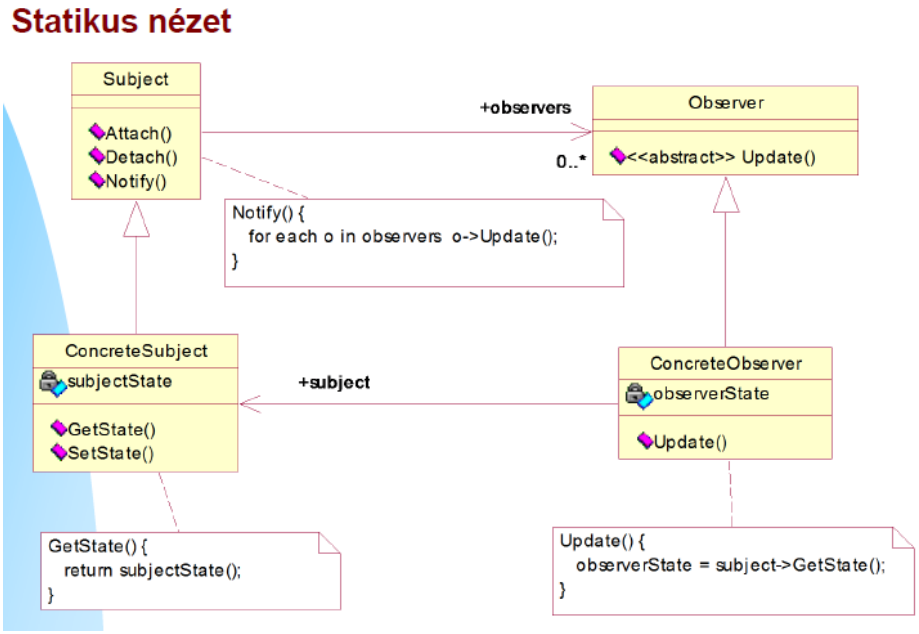
- ☞ **megoldás:** programozási nyelvek segítségét igényli (**static**). Az osztály felelőssége, hogy egy példány jöhessen csak létre belőle, továbbá, hogy ahhoz globális hozzáférést biztosítson.

```
public class Singleton
 {
     private static Singleton instance = null;
     public static Singleton Instance
     {
        get
        {
           if (instance == null)
               instance = new Singleton();
           return instance;
         }
     }

     protected Singleton() { }
     public void Print() {…}
}


Használata:

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

Strukturális és viselkedési tervezési minták:

➢ Observer:

- **probléma**: hogyan tudják objektumok értesíteni egymást állapotuk megváltoztatásáról anélkül, hogy bármit is tudna róluk (pl. a modellben ne legyen hivatkozás a megjelenítési osztályra – újrafelhasználhatóság)
- **megoldás**: az MVC vagy Document-View architektúra: **azaz a modell csak egy közös interfészen (*Observer*) keresztül tárolja a beregisztrált view-kat.**
    - emeljük ki az adatokat és a rajta értelmezett műveleteket egy modell osztályba
    - a modellhez view-kat lehet beregisztrálni (observers lista)
    - a modell értesíti a beregisztrált view-kat,
    - és az értesítés hatására minden view lekérdezi a modell állapotát és az alapján frissíti magát

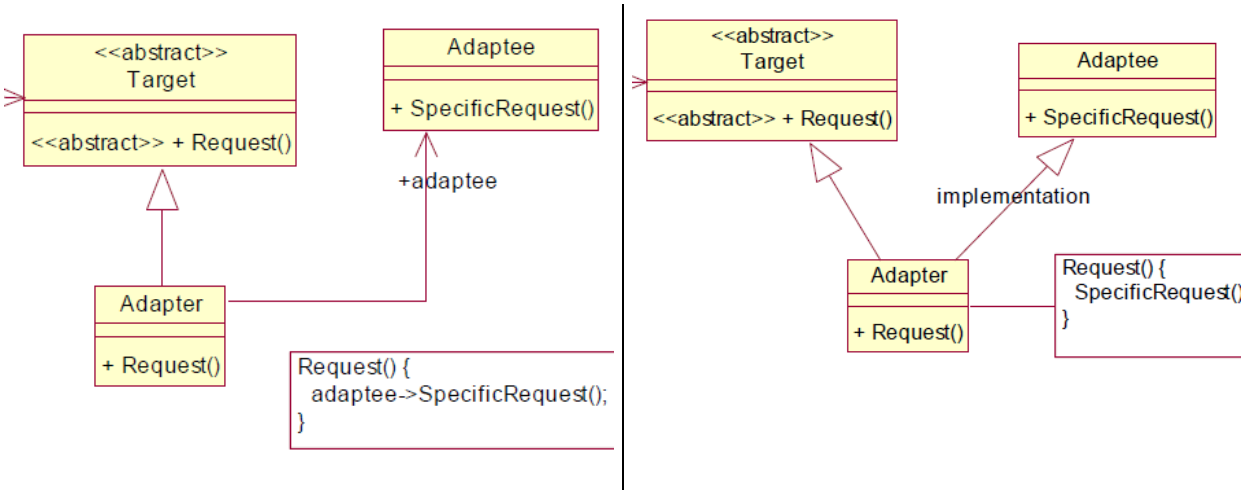


➢ Adapter (wrapper):

- **probléma:** Lehetővé akarjuk tenni olyan osztályok együttműködését, amik egyébként az inkompatibilis interfészeik miatt nem tudnak együttműködni.
  pl: grafikus alakzatok a Shape osztályból származnak, többek között meg akarunk valósítani egy TextShape osztályt. Ennek megírás nehéz, azonban a framework amin dolgozunk biztosít egy TextView osztályt ami tudja mindazt, amit a TextShape-től elvárunk. A probléma, hogy interfésze nem kompatibilis a Shape osztály interfészével (nem a Shape-ből származik).
- **megoldás:** object adapter vagy class adapter:
  Szereplők: Target (Shape), Adapter (TextShape), Adaptee (TextView).

  Object adapter esetén az Adapter tagváltozóként tartalmaz egy Adaptee referenciát, a műveletek végrehajtását ezen a referencián keresztül delegálja.

  Class adapter esetén az Adapter nem csak a Targetből származik le, hanem az Adapteeből is (többszörös öröklés).

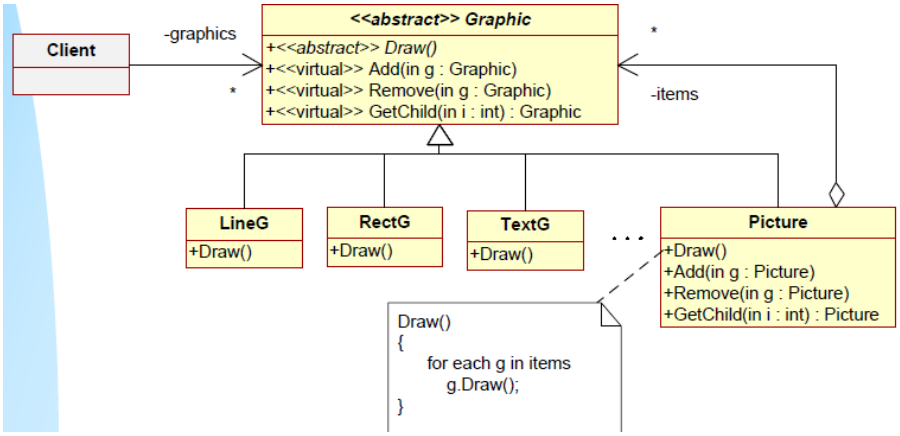


object adapter

class adapter

Composite:

- **probléma:** objektumok fa struktúrába rendezése. A kliensek számára el akarjuk rejteni, hogy egy objektum valójában kompozit objektum (egységesen szeretnénk kezelni őket).
  pl.: olyan grafikus alkalmazás, ami lehetővé teszi összetett grafikus objektumok létrehozását.
- **megoldás:** composite tervezési minta:

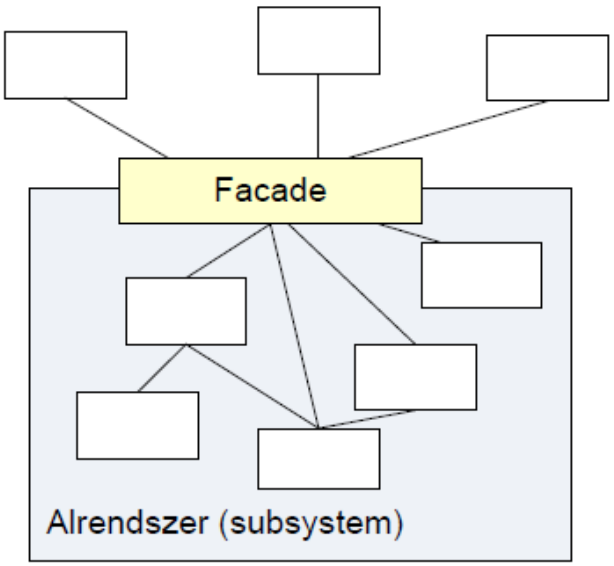


Facade:

- **célja:** egységes (magasabb szintű) interfészt definiálni egy alrendszer interfészeinek egyszerűbb használatához.

- Ilyen pl. a vállalati rendszerek háromrétegű architektúrájában megismert üzleti logika (tartomány). A megjelenítési réteg csak a vékony üzleti logikai réteghez tartozó interfész osztályait látja.

- használjuk ha: egyszerű interfészt szeretnénk biztosítani egy komplex rendszer felé.

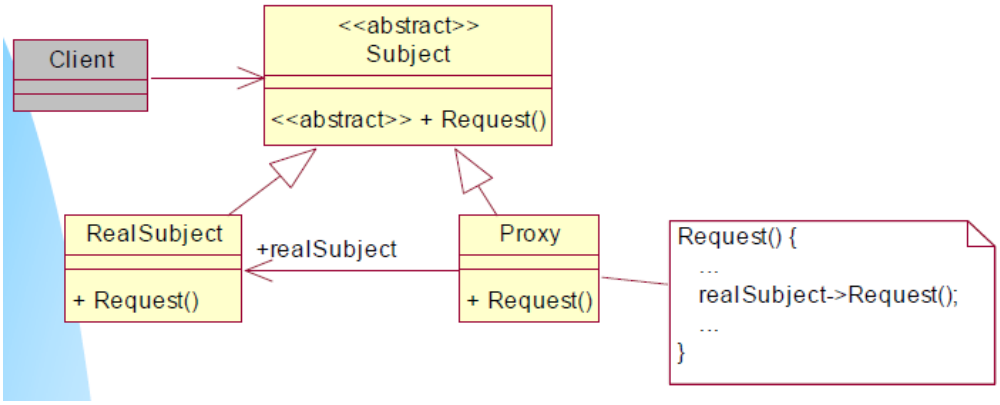


➢ Proxy:

- ☞ **probléma**: az objektumhoz való hozzáférés szabályozása
- ☞ **megoldás**: az objektum helyett egy helyettesítő objektumot használ, ami szabályozza az objektumhoz való hozzáférést.

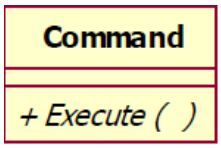
Pl: nagyméretű képek, nem kell őket betölteni egyszerre, csak ha odagörgetünk. Helyettesítsük a képeket egy proxyval, ami megoldja, hogy ha odagörgetünk, és a kép már be van töltve akkor megjeleníti, különben betölti, majd megjeleníti.

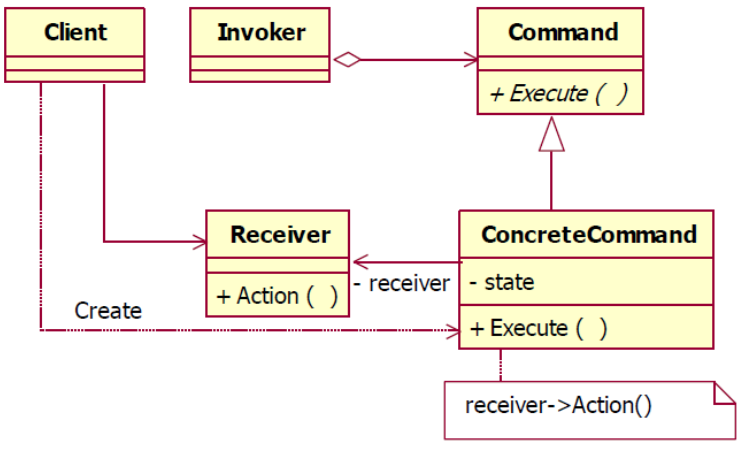
- ☞ szereplők: Subject, RealSubject, Proxy



➢ Command:

- ☞ egy tipikus felhasználási területe a GUI keretrendszerre való fejlesztés
- ☞ **probléma:** a keretrendszer készítői nem építhették bele az egyes menüelemek kiválasztásának kezelését (mert az alkalmazásfüggő). Hogyan kezeljük ezeket? Visszavonás?
- ☞ **megoldás:** OO környezetben command mintára építve. Minden felhasználói utasításnak (pl. save) feleltessünk meg egy *Command* leszármazott objektumot. A falhasználói utasítás hatására ennek az objektumnak az Execute() metódusát hívjuk meg.



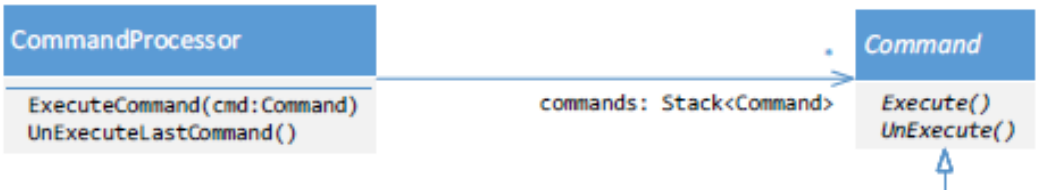


Példában:
Invoker – Menu/MenuItem
Reciever – ált. az App

- ☞ **előnye:**
  - elválasztja a parancsot kiadó objektumot attól, amelyik tudja, hogy hogyan kell azt kezelni – egységbezárás, bővíthetőség
  - ugyanazon parancs több GUI elemhez is hozzárendelhető (tipikusan menüelem, toolbar gomb)
  - visszavonás támogatása (Command Processor segítségével)

☞ **Command Processor:**

- a command minta egy változata
- a Command osztályt kiegészítjük egy UnExecute() metódussal
- létrehozunk egy CommandProcessor objektumot, ennek feladata:
  - a végrehajtott Command-ok tárolása – command-stack (így visszavonható)
  - ExecuteCommand(Command c): végrehajtja a paraméterként kapott utasítást
  - UnExecuteLastCommand(): kiveszi az utoljára végrehajtott utasítást a command-stackből és meghívja annak az UnExecute() műveletét.



➢ Memento:

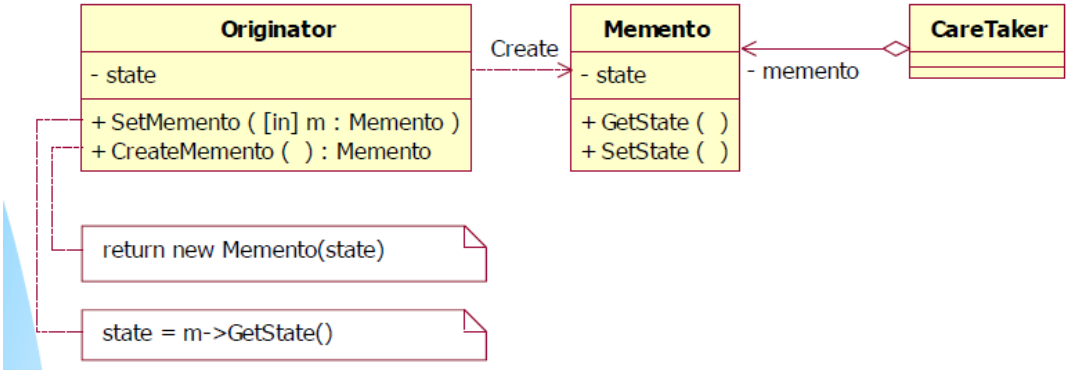
☞ **probléma:** általában visszavonás funkció megvalósítása során merül fel. Ez gyakran lehetetlen anélkül, hogy az objektum teljes állapotát elmentenénk, majd visszaállítanánk – az objektum teljes állapota azonban nem elérhető, mert a tagváltozók private-ok.

☞ **megoldás:** a memento minta. Egy objektum egy adott állapotát egy Memento osztályba csomagoljuk, és ilyen „becsomagolt” formában tesszük elérhetővé.

☞ hátrány: erőforrásigényes

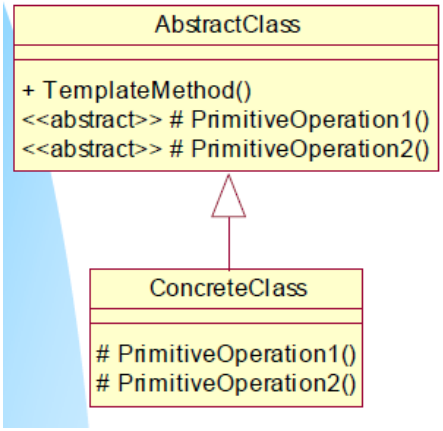
☞ szereplők:

- Originator – az ő állapotát (-state) kell tudni visszaállítani
- Memento – az állapotot tárolja
- CareTaker – nyilvántartja a Memento-kat



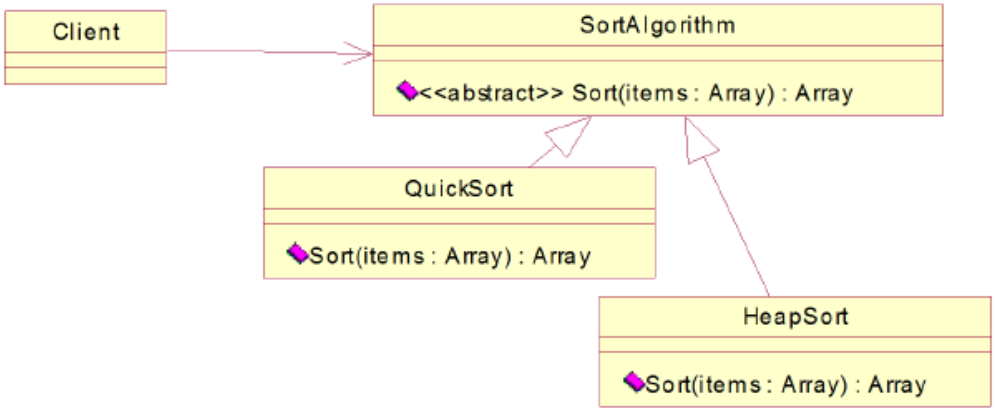
Template method:

- **lényege:** egy algoritmusvázat definiál, és ennek a váznak néhány metódusimplementációját a leszármazott osztályokra bízza.
- **felhasználása pl.:** framework esetén adott egy váz, amire építeni lehet, azonban az alkalmazás specifikus megoldásokat a programozó valósítja meg azzal, hogy az absztrakt metódusvázakat megvalósítja.



Strategy:

- **probléma:** az algoritmusok egységbe zárása. A kliens szemszögéből az általa használt algoritmusok szabadon kicserélhetőek legyenek.
  pl.: a kliens objektum többféle sorrendező algoritmust használhat
- **megoldás:** A kliens tartalmaz egy SortAlgorithm referenciát egy konkrét QuickSort vagy HeapSort objektumra. A hivatkozott implementációs objektum könnyen cserélhető.



Strategized locking:

- **lényege:** ez tulajdonképpen a Strategy minta alkalmazása. A fenti példában rendező algoritmusok egyszerű cserélhetőségét valósítottuk meg, itt egy adott osztályt kezelő algoritmus szálbiztos / egyszálú környezetben alkalmazott algoritmusa között válthatunk az implementációs objektum cseréjével.
- **a minta leírása:**
  - bevezetünk egy *ILock* interfészt, Acquire() és Release() metódusokkal
  - ehhez írunk két féle implementációt:
    - NoLock: az Aquire() és Release() metódusok nem csinálnak semmit
    - ObjectLock: az Aquire() objektum szinten zárol, a Release() oldja a zárat
  - a védendő osztályunkba felveszünk egy ILock referenciát, amit egyszálú környezetben egy NoLock, többszálú környezetben egy ObjectLock referenciára állítunk

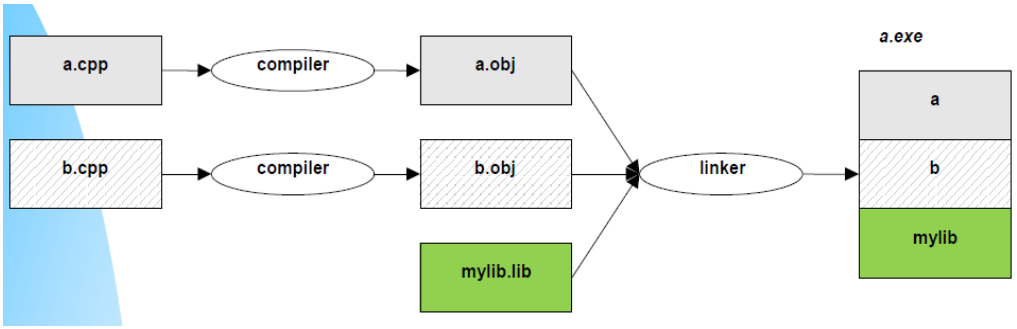
Bináris komponensek, reflexió (ea. 13):

Fordítás folyamata (C, C++ környezetben): a forrásfájlokból a compiler .o vagy .obj fájlokat készít, ezeken a linker címfeloldást végez, így áll elő a kimeneti .exe fájl.

Könyvtár: függvénydefiníciókat, osztálydefiníciókat, konstansokat, lefordított (bináris) erőforrásokat tartalmaz. Ezeket **bináris komponens**eknek nevezzük.
Ezeket includeolni, linkelni kell. Linkelés két módja: **statikus / dinamikus linkelés**

Statikus linkelés:

- A könyvtár .lib kiterjesztésű, ez tulajdonképpen több .obj fájl összefűzve.



- Hátrányok:
  - nagy fájlméret az alkalmazásnál – ahány .exe-hez linkelem, annyiszor foglal helyet a háttértáron.
  - nagy memóriaigény
  - nehéz hibajavítás
- Előnyök:
  - önállóan futni képes alkalmazás, nincs szükség külső könyvtárfájlokra

Dinamikus linkelés:

- A könyvtár Windows alatt .dll, Linux alatt .so kiterjesztésű.
- A **futó** program csak **futásidőben tölti be a számára szükséges könyvtárakat**.

- Hátrányok:
  - a futtatáskor jelen kell hogy legyenek a használt .dll -ek
  - DLL hell probléma

- Előnyök:
  - kisebb háttértárigény – csak egyszer foglal helyet (pl.: windows\system könyvtárban)
  - az alkalmazás mérete is kisebb
  - kevesebb memóriaigény - ha több folyamat is használja ugyanazt a könyvtárat, akkor a memóriába csak egy példányba töltődik be
  - egyszerűbb hibajavítás – elég a .dll cseréje

- **A DLL kódja csak egyszer töltődik be a memóriába, a DLL adat terület viszont annyiszor, ahány folyamat használja adott pillanatban a DLL-t.** Csak így lehetnek védettek egymással szemben a folyamatok.

Bináris komponensek:

C++ környezetben ha egy adott komponens (pl. .exe fájl) használ egy .dll fájlt, akkor a .dll fájl belső szerkezetének módosítása (pl. felveszünk egy privát tagváltozót) esetén is az őt használó komponenst újra kell fordítani. Ez megnehezíti a többkomponensű alkalmazások fejlesztését.

.NET környezetben a .dll fájlt használó komponenst mindaddig nem kell újra fordítani, amíg a .dll-en végzett változtatások nem érintik annak interfészét. Ez azért lehetséges, mert .NET környezetben a komponensek nem függenek a más komponensekben lévő osztályok belső szerkezetétől.

Reflexió:

Elsősorban **szerelvények és típusok meta-adatait kérdezhetjük le.**

Fontosabb szolgáltatásai:

- egy objektum/osztály típusának lekérdezése:

```
Complex c1 = new Complex(10, 10);
Type t1 = c1.GetType(); // A GetType az Object-ben definiált
Console.WriteLine(t1.FullName);
// A typeof egy C# operátor
Type t2 = typeof(Complex);
Console.WriteLine(t2.FullName);
```

```
file:///C:/Data/Tanszek/O
MiscDemos.Complex
MiscDemos.Complex
```

- egy szerelvényben lévő típusok lekérdezése:

```
Assembly assembly;
assembly = Assembly.LoadFrom("MyLib.dll");
Type[] types = assembly.GetTypes();
foreach (Type type in types)
   Console.WriteLine(type.FullName);
```

```
file:///C:/Data/Tansz
MyLib.Person
MyLib.Complex
```

- osztály tagváltozóinak és tagfüggvényeinek lekérdezése:

```
Type type = typeof (Complex);
Console.WriteLine("Fields:");
foreach (FieldInfo fi in type.GetFields())
    Console.WriteLine("\t" + fi.Name);
Console.WriteLine("Methods:");
foreach (MethodInfo mi in type.GetMethods())
    Console.WriteLine("\t" + mi.Name);
```

```
file:///C:/Data/Tanszek/Okta
Fields:
        Re
        Im
Methods:
        Print
        ToString
        Equals
        GetHashCode
        GetType
```