Tervezési minták (design patterns)
- strukturális minták
- viselkedési minták

Szoftvertechnikák 16-17. előadás

Benedek Zoltán
Automatizálási és Alkalmazott Informatikai Tanszék, BME

Adapter
Bridge
Composite
Decorator
Facade
Proxy

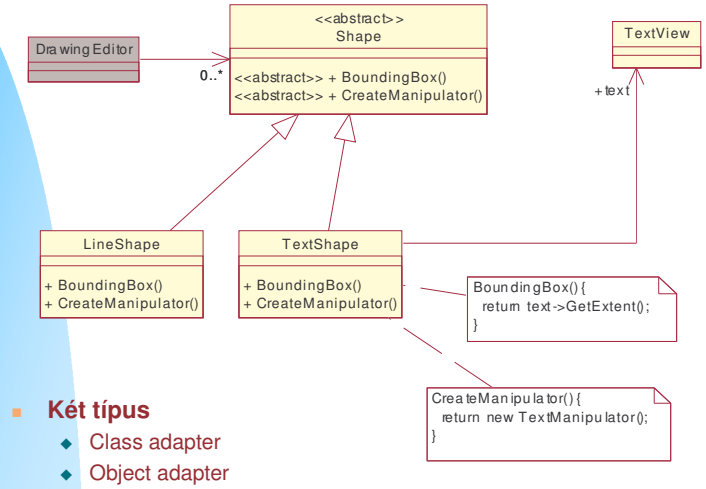
STRUKTURÁLIS MINTÁK

Adapter (wrapper)

Adapter

- **Cél**
  - Egy osztály interfészét olyan interfésszé konvertálja, amit a kliens vár. Lehetővé teszi olyan osztályok együttműködését, melyek egyébként az inkompatibilis interfészeik miatt nem tudnának együttműködni.
- **Példa**
  - Grafikus Editor
    - Grafikus alakzatok (vonalak, poligon, **szöveg**) – a Shape osztályból származnak (vonal – LineShape, poligon – PoligonShape, szöveg – TextShape, stb)
    - TextShape megírása nehéz, viszont tegyük fel, hogy a framework egy sokoldalú TextView osztály, ami mindazt, tudja, amit a TextShape-től elvárunk
    - Probléma: a TextView osztályt nem tudjuk közvetlenül felhasználni, mert nem megfelelő az interfésze, ugyanis nem a Shape osztályból származik (nem támogatja a Shape interfészt, emiatt nem tudjuk a többi Shape-el együtt egységesen kezelni)
    - Megoldás: Adapter minta használata

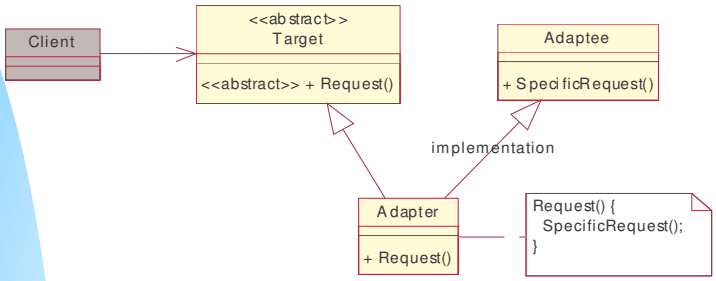
Adapter



- **Két típus**
  - Class adapter
  - Object adapter
- **A fenti példa egy „object” adapter**

Adapter

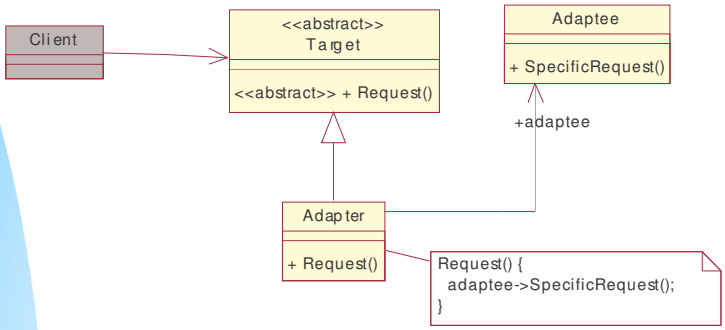
- **Struktúra – class adapter (első verzió)**



- Többszörös örökléssel oldja meg az adaptálást
- *Szereplők*
  - *Adaptee (TextView):* interfész, amit adaptálni (illeszteni) kell
  - *Adapter (TextShape) :* illesztő, az Adaptee interfészt a Target interfésszé konvertálja
  - *Target (Shape):* interfész, amit a kliens használ

Adapter

- **Struktúra – object adapter (második verzió)**



- Objektum kompozícióval, delegálással oldja meg az adaptálást
- Szereplők: mint a *Class Adapter*-nél
  - Az Adapter tartalmaz egy pointert vagy referenciát az Adaptee-ra
  - Az adapter delegálja a művelet végrehajtását az Adaptee-ra
  - Egy adapter képes több Adaptee-t is magában foglalni, beleértve azok alosztályait is

Adapter

- **Implementáció**
  - C++-ban a class adapter esetén
    - Private öröklés az Adaptee-ból (implementáció öröklés)
    - Public öröklés a Target-ból (interfész öröklés)

Adapter

- **Használjuk, ha**
  - Egy olyan osztályt szeretnénk használni, amelynek interfésze nem megfelelő
  - Egy újrafelhasználható osztályt szeretnénk készíteni, amely együttműködik előre nem látható vagy független szerkezetű osztályokkal (pluggable adapters, lásd rövidesen)

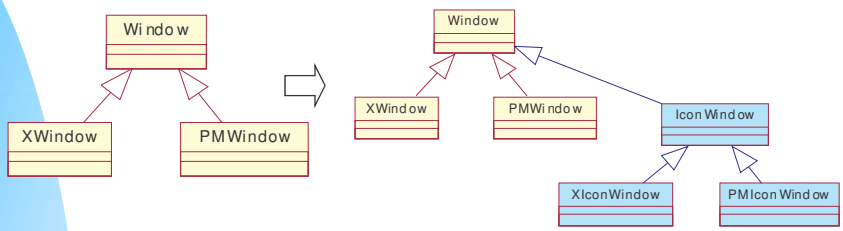
Bridge

Bridge

- **Cél**
  - Különválasztja az absztrakciót (interfészt) az implementációtól, hogy egymástól függetlenül lehessen őket változtatni
- **Példa**
  - Példa: hordozható ablakozós rendszer XWindow és Presentation Manager alá
- **Megoldás**
  - A, Leszármaztatással, pl. minden ablak implementációnak egy ***Window*** osztály leszármazott (lásd rövidesen)
    - Inkonzisztens hierarchia
    - Platformfüggő lesz a kliens
  - B, Bridge pattern alkalmazása (lásd rövidesen)

Bridge

A, Megoldás leszármaztatással



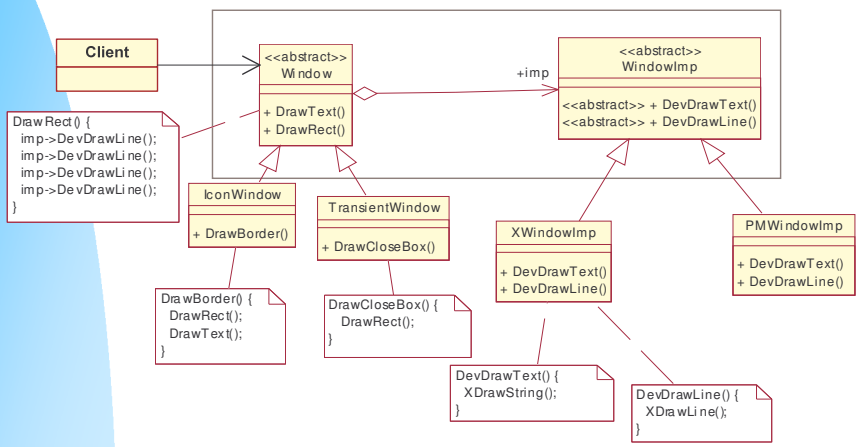
Problémák a leszármaztatással

1. Inkonzisztens hierarchia. Ez új ablaktípus bevezetésénél (a példánkban ***IconWindow***) jön elő: az ikonos ablaknak XWindow és Presentation Manager alá is kell implementációt készíteni. Minden platform (X, PM, stb.) - ablaktípus (Icon, sclollozható, stb.) kombinációra kell implementációt készíteni! Ez így használhatatlan!
2. Probléma a leszármaztatással: ha a kliens kódjában benne lesz, hogy XWindow, XIconWindow (ebből hoz létre példányokat), akkor csak az X platformon fog futni, nem lesz hordozható a kód

Megoldás: használjuk a Bridge patternt →

Bridge

- **Bridge pattern**



Bridge

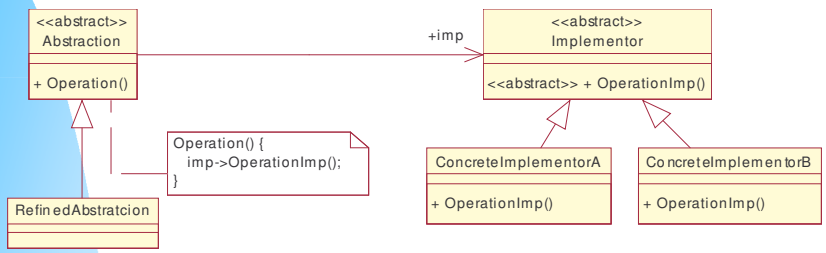
- **Megoldás**
  - A ***Window*** absztrakció és implementáció külön hierarchiába rendezése. Külön hierarchiába:
    - ***Window*** interfészek
    - Platform specifikus ***WindowImp*** implementációk
  - A ***Window*** osztály tartalmaz (hivatkozik) egy implementáló objektumot, a *Window* műveletei a funkcióikat ennek segítségével valósítják meg
- **Bridge használatának előnyei**
  - Az absztrakció és az implementáció különválasztása
    - A példában a ***Window*** osztály és leszármazottainak a műveletei az applikáció fejlesztő világképét követi, az interfészei az alkalmazásfejlesztőt szolgálhatják. A ***WindowImp*** osztály és leszármazottainak a műveletei az ablakozó rendszer világképét követik: az ablakozós rendszerek fejlesztőit szolgálják
    - Így az absztrakciós és az implementációs hierarchia egymástól függetlenül fejleszthető, bővíthető. A példánkban az egy új ablaktípus bevezetéséhez csak az absztrakciós hierarchiát kell bővíteni (új ***Window*** leszármazott), az automatikusan minden platformon támogatva lesz. Vagy: a példánkban egy új platform (pl. Linux támogatásához csak az implementációs hierarchiát kell bővíteni (új ***WindowImpl*** leszármazott), minden ablaktípus automatikusan támogatva lesz..

Bridge

- **Bridge használatának előnyei**
  - Az implementáció dinamikusan, akár futási időben is megváltoztatható (ehhez mindössze a ***Window*** osztály mutatóját/referenciáját kell egy másik implementációs objektumra állítani.
  - Az implementációs részletek a klienstől teljesen elrejthetők. Sőt: az implementációs osztálynak még az interfésze sem látszik a kliens számára! Ez az UML ábrán jól látszik.
    - Vagyis a bridge alkalmazásával elérhető, hogy egy osztály **interfészének** megváltozása (a példánkban a ***WindowImpl***) ne érintse a (közvetett) felhasználó osztályt (a példánkban a ***Client***)
    - Vagyis a bridge alkalmazásával elrejthetjük interfészeinket, ha annak kialakítása komoly szellemi értéket képvisel
    - Vagyis a bridge alkalmazásával elérhetjük, hogy az implementációs interfészeink a jövőben szabadon megváltoztathatók legyenek
  - Az implementációs hierarchia külön lefordított komponensbe tehető (C++-nál .lib, .dll, .NET esetében dll), így ha ez ritkán változik, nagy projektek esetén nagymértékben gyorsítható a fordítás/buildelés ideje.
  - Ugyanaz az implementációs objektum, több helyen is felhasználható (pl. referencia számlálással)

Bridge

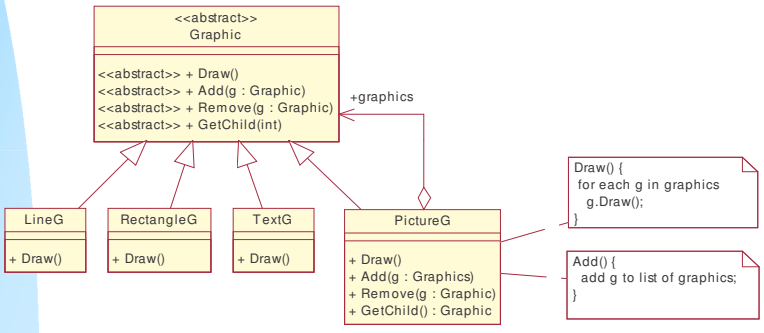
- **Bridge pattern struktúra**



Composite

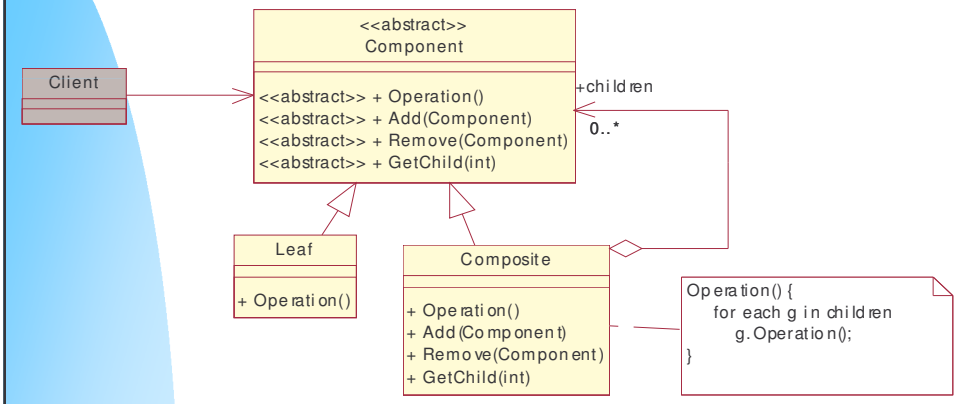
Composite

- **Célja**
  - Rész-egész viszonyban álló objektumokat fastruktúrába rendezi
  - A kliensek számára lehetővé teszi, hogy az egyszerű és kompozit objektumokat egységesen kezelje
- **Példa**
  - Olyan grafikus alkalmazás, amely lehetővé teszi összetett grafikus objektumok létrehozását



Composite

- **Struktúra**



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.
- **Megjegyzés:** Kérdés, hogy melyik osztályban definiáljuk a kompozit kezelő műveleteket (Add, Remove, ...), amelyek nem értelmezhetők a levél objektumokra (LineG, TextG, stb)
  - A "**omponent** osztályban (ez volt a példában):
    - előny, hogy egységesen kezelhető minden objektum (kompozit és nem kompozit objektumok), hiszen mind támogatja az Add, Remove, GetChild műveleteket
    - hátránya, hogy nem biztonságos, de megoldás lehet pl. Exception dobása, ha a művelet nem értelmezhető
  - A ***Composite*** osztályban:
    - Hátrány: fordítva, mint az előbb
    - Pl. megoldás arra, hogy a kliens el tudja dönteni, hogy egy objektum kompozit-e: **IsComposite***: bool* virtuális függvény definiálása a **Component** osztályban, a levél osztályokban false-al térjen vissza.

Composite

- **Kérdés:** Hol jelenik meg a Composite minta a Windows Forms alkalmazásokban?
- **Feladat1:**
  - Egy 3D-s CAD program esetén összetett, sokkomponensű térbeli testek térfogatát kell kiszámítani. Milyen módon segít a megoldásban a *Composite* minta?

Decorator (toldalék)

Decorator

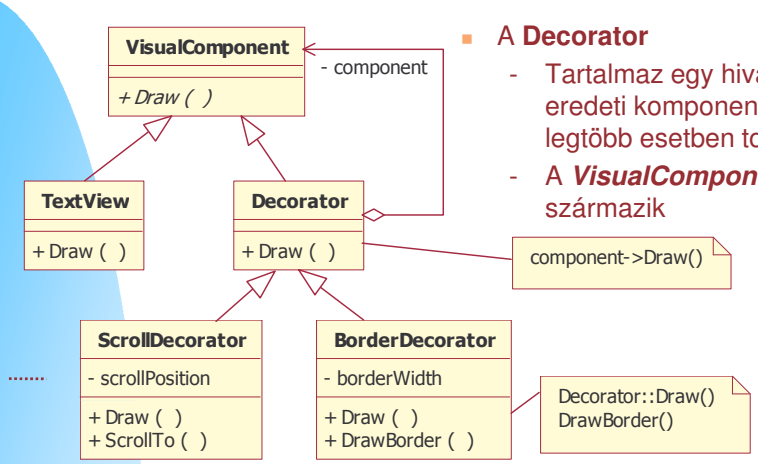
- **Célja**
  - Objektumok funkciójának dinamikus kiterjesztése
  - Rugalmas alternatívája a leszármaztatásnak
- **Példa: adott egy GUI keretrendszer (pl. Windows Forms-hoz, AWT-hez hasonló)**
  - Az ablakokokhoz, vezérlőelemekhez hozzá szeretnénk rendelni
    - Keretet (Border)
    - Görgetősávot (Scrollbar)
    - Animációt (Animation) – pl. ha az egér fölé ér
    - Árnyékot - pl. ha az egér fölé ér
    - Stb.
  - Ezeket tetszőleges kombinációban szeretnénk az osztályokhoz rendelni
  - Tegyük fel hogy beépítve NEM támogatja a keretrendszer !!!

Mit tehetünk?

Decorator

- **Megoldások**
  - A, Öröklés
    - Nem rugalmas – minden elemhez hozzá lesz rendelve és a kliens nem tudja ezt szabályozni futási időben
      - Minden vezérlőelemhez hozzá lesz rendelve a Keret, Görgetősáv, Animáció, stb. attól függetlenül, hogy szükség van-e rá.
    - Nem kombinálhatók tetszőleges módon az új szolgáltatások (csak ha nagyon sok leszármazottat hozunk létre (pl. BorderedScrollableTextView, stb.), vagy egy leszármazottba minden funkciót belepakolunk. Ez utóbbi a gyakorlatben ritkán probléma!
    - Sokszor nem szerencsés leszármaztatni (.NET, Java: csak egy ősosztály lehet). Ez ritkán probléma.
  - B, Decorator minta alkalmazása
    - Csomagoljuk be az objektumot egy dekorátor objektummal
    - A csomagoló objektum nyújtja a plusz szolgáltatásokat (scroll, border, animation, shadow)
    - Transzparens módon: ugyanaz az interfésze, mint az eredeti objektumnak, így a kliens számára átlátszó, másrészt rekurzívan alkalmazhatók a dekorátor objektumok!

Decorator



- A **Decorator**
  - Tartalmaz egy hivatkozást az eredeti komponensre, a legtöbb esetben továbbhív
  - A ***VisualComponent***-ből származik

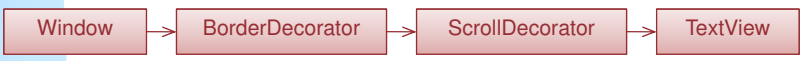
- ***Decorator*** objektumok mindenhol szerepelhetnek, ahol ***VisualComponent*** szerepelhet: a kliens meg sem tudja különböztetni → a GUI keretrendszert nem kell (nem is lehet!) megváltoztatni.

Decorator

- **Pszeudo kódrészlet a felkonfiguráláshoz**

```
// Az ablak vezérlőeelmeinek inicializálása
public class MyForm: Form
{
   public void InitComponents()
   {
       TextView textView = new TextView();
       // A Controls.Add VisualComponent-tet vár. A
       // Decorator-ok is ebből származnak, így megadhatók
       // paraméterként.
       this.Controls.Add(
         new BorderDecorator(
            new ScrollDecorator(textView), 1 ) );
   }
}
```

- **Objektumok egymásra hivatkozása a fenti példában**



Decorator

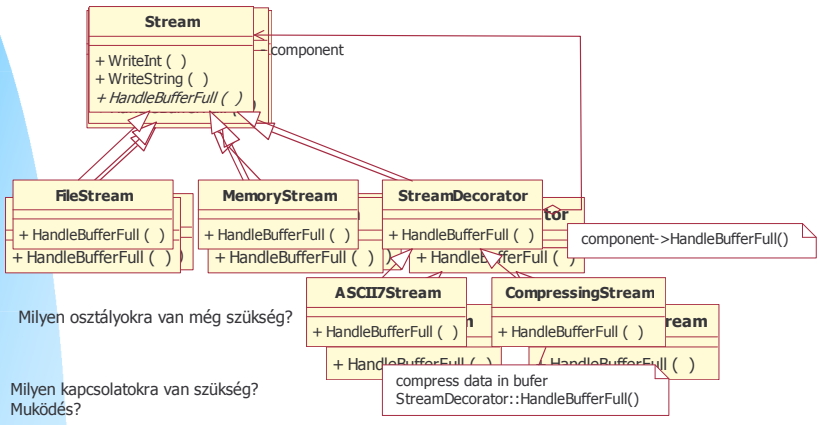
- **Használjuk, ha**
  - Dinamikusan szeretnénk funkcionalitást/viselkedést hozzárendelni az egyes objektumokhoz
  - A funkcionalitást a kliens számára átlátszó módon szeretnénk az objektumhoz rendelni
  - Amikor a származtatás nem praktikus

- **Előnyök**
  - Sokkal rugalmasabb, mint a statikus öröklődés
  - Több testreszabható osztály határozza meg a tulajdonságokat

- **Hátrányok**
  - Némiképp bonyolultabb, mint az egyszerű öröklés (több osztály szerepel)
  - A decorator és a dekorált komponens interfésze ugyan azonos, de maga az osztály nem ugyanaz. Ha reflexióval építünk a konkrét típusra, akkor a dekorátor alkalmazása problémát okozhat.

Decorator

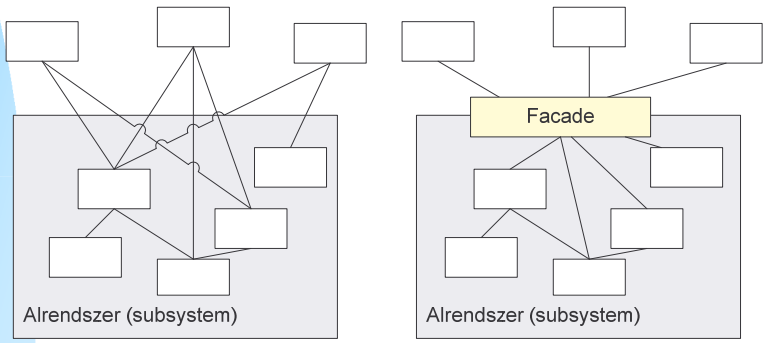
Feladat : Stream-ek kezelése. Tervezzük meg úgy, hogy tudjon tömöríteni, illetve ASCII-be konvertálni tetszőleges kombinációban.



Facade

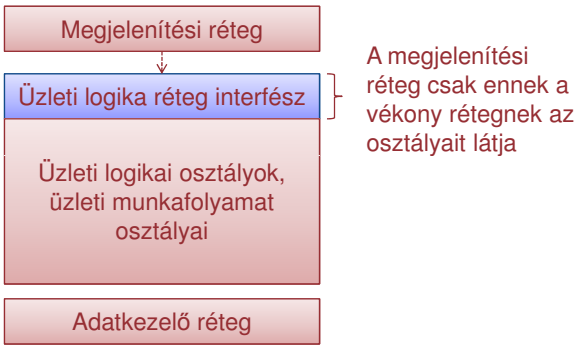
Facade

- **Célja**
  - Egységes interfészt definiál egy alrendszer interfészeinek halmazához
  - Magasabb szintű interfészt definiál, amin keresztül könnyebb az alrendszer használata
- **Példa**



Facade

- **Példa 1**
  - Compiler
    - Command line hívható cpp.exe, nagyon sok argumentummal
    - A belsejéhez nem férünk hozzá (parser, tokenizer, stb.)
- **Példa 2**
  - Többrétegű (többnyire üzleti) alkalmazások



Facade

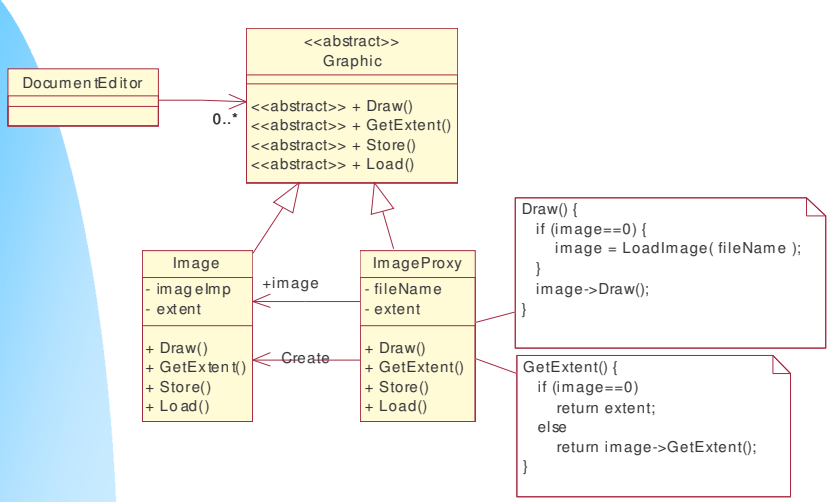
- **Használjuk, ha**
  - Egyszerű interfészt szeretnénk biztosítani egy komplex rendszer felé
  - Számos függőség van a kliens és az alrendszerek osztályai között. Ilyenkor ha létrehozva egy Facade-ot elősegítve az alrendszer függetlenségét és a hordozhatóságot
  - Layers (rétegelés) esetén
- **Megjegyzés**
  - Külön döntés, hogy engedünk-e hozzáférést az alrendszerek osztályaihoz
    - Elvileg nem akadályozza meg, hogy a kliensek felhasználják az alrendszerek osztályait, ha arra is szükségük van

Proxy

Proxy

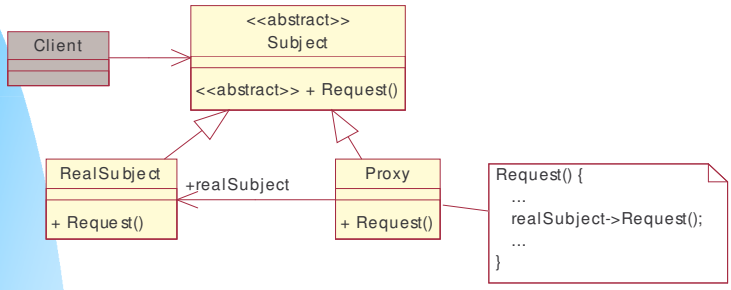
- **Célja**
  - Objektum helyett egy helyettesítő objektumot használ, ami szabályozza az objektumhoz való hozzáférést
- **Példa**
  - Szövegszerkesztő
    - Sok nagy méretű kép
    - Nem kel őket egyszerre megjeleníteni, csak amikor odagörgetjük az ablakot, vagyis amikor láthatóvá válik
  - Megoldás: Proxy
    - Helyettesítsük a képet egy objektummal (proxy), amely ha be van töltve a kép megjeleníti, egyébként betölti, és azután jeleníti meg

Proxy



Proxy

Struktúra



- ***Subject***: közös interfészt biztosít a Subject és a Proxy számára (ezáltal tud a minta működni, ez a lényeg!)
- ***Realsubject***: a valódi objektum, amit a proxy elrejt
- ***Proxy***: helyettesítő objektum. Tartalmaz egy referenciát a tényleges objektumra, hogy el tudja azt érni. Szabályozza a hozzáférést a tényleges objektumhoz, feladata lehet a tényleges objektum létrehozása és törlése is.

Proxy

- **Távoli Proxy**
  - Távoli objektumok lokális megjelenítése átlátszó módon. A kliens nem is érzékeli, hogy a tényleges objektum egy másik címtartományban, vagy egy másik gépen van
- **Virtuális Proxy**
  - Nagy erőforrás igényű objektumok igény szerinti létrehozása (pl. kép)
- **Védelmi Proxy**
  - A hozzáférést szabályozza különböző jogok esetén
- **Smart Pointer**
  - Egy pointer egységbezárása, hogy bizonyos esetekben automatikus műveleteket hajtson végre (pl.:lockolás)

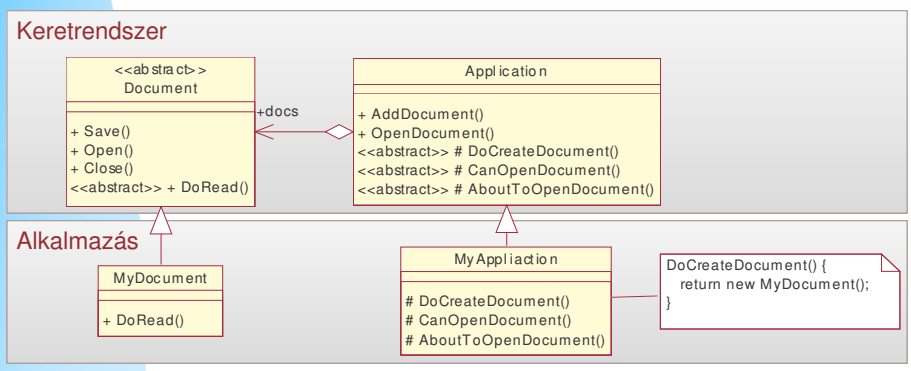
Template Method
Iterator
Observer
Mediator
Chain of Responsibility
Command
Memento
State
Strategy
Visitor
(Iterpreter)

VISELKEDÉSI MINTÁK

Template method

Template method

- **Cél**
  - Egy műveleten belül algoritmus vázat definiál, és ennek néhány lépésének implementálását a leszármazott osztályra bízza.
- **Példa: Framework-ben dokumentum megnyitása**
  - A framework-ben legyen adott két osztály, ***Application*** és ***Document***. Ezekből kell a programozónak egy-egy saját osztály leszármaztatnia, amikben megvalósítja az alkalmazásspecifikus viselkedést.



Template method

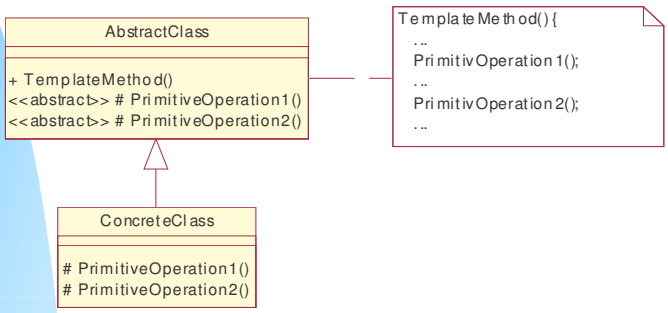
- **A példában az OpenDocument egy ún. template method**
  - Meghatározza a műveletek sorrendjét
  - Meghív néhány absztrakt műveletet, melyeket a leszármazott osztályban felül kell definiálni, hogy meghatározott viselkedést rendeljünk hozzá az aktuális igényeknek megfelelően

```
// Az Application osztály a framework része
void Application::OpenDocument (const char* name) {
   if (!CanOpenDocument(name)) {
       // cannot handle this document
       return;
    }
    // a DoCreateDocument-et a adott alkalmazás megírásakor az
    // az Application-ból leszármazott MyApplication osztályban
    // felül kell definiálni, itt lesz majd értelmesen „kitöltve”
    Document* doc = DoCreateDocument();

    if (doc) {
            _docs->AddDocument(doc);
            AboutToOpenDocument(doc);
            doc->Open();
            doc->DoRead();
    }
}
```

Template method

- **Struktúra**



Template method

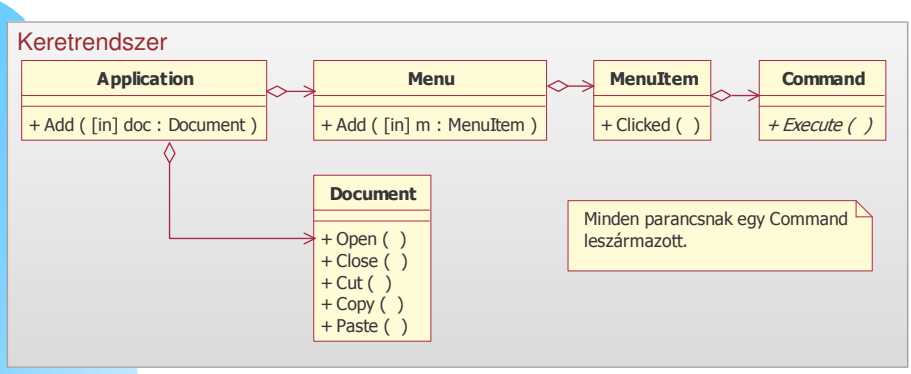
- **Következmények**
  - Lehetővé teszi, hogy az algoritmus invariáns részeit egy helyen definiáljuk, és a változó részeket a leszármazott osztályban adjuk meg. Így megoldható a kódduplikálás elkerülése: a hierarchiában a közös kódrészeket a szülő osztályban egy helyen adjuk meg (template method), ami a különböző viselkedést megvalósító egyéb műveleteket hívja meg. Ezeket a "különböző viselkedést megvalósító egyéb műveleteket" a leszármazott osztályban felül kell/lehet definiálni.
  - Lehetővé teszi ún. hook függvények definiálását: ezek kiterjesztési pontok a kódban.
- **Megjegyzés**
  - Kretrendszerek esetében gyakori
  - A .NET-ben delegate-tel is megoldható a kiterjesztés, C++, Java esetén csak az itt bemutatott (ősben levő virtuális függvény hívása) lehet megoldás
  - Semmi köze a C++ sablonokhoz (azok generikus típusok)

Command

Command

- **Cél**
  - Egy kérés objektumként való egységbezárása. Ez lehetővé teszi a kliens különböző kérésekkel való felparaméterezését, a kérések sorba állítását, naplózását és visszavonását (undo)
- **Alternatív nevek**
  - Action
- **Nagyon rendszerfüggő (C++, .NET, stb.) a koncepció és az implementáció is**
- **Példa: felhasználói parancsok**
  - Menü, gomb, toolbar gomb
  - Résztvevők pl.: Alkalmazás, Dokumentum, Menü, Almenü, Menüpont, stb
  - Probléma: a GUI keretrendszer írói nem építhették bele az alkalmazásfüggő menüelem kiválasztás kezelést →
  - Hogyan reagáljunk a menüpont kiválasztása által generált eseményekre?
    - Callback függvény – nem objektum-orientált (strukturált) megoldás
    - Adapter alapú megoldások – Java
    - Delegate alapú megoldás - .NET
    - Command minta

Command

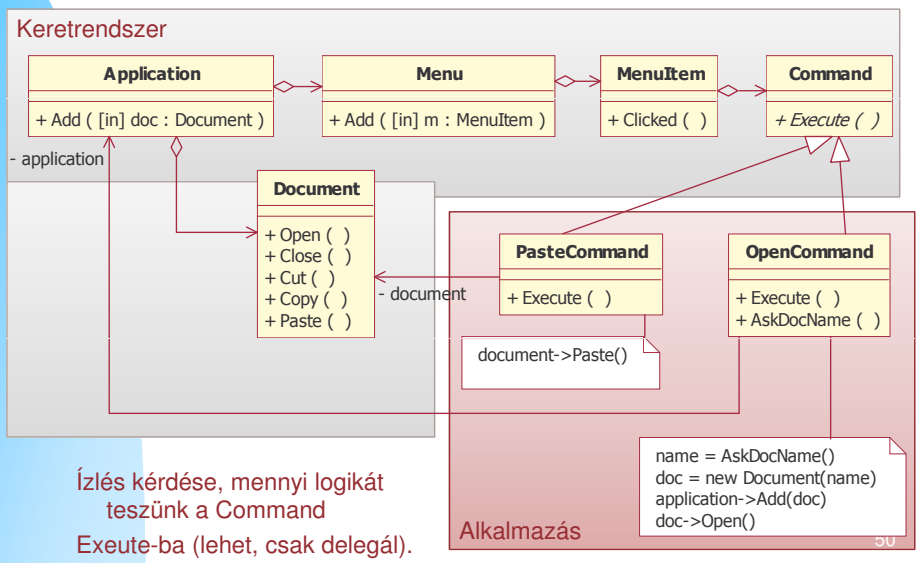


- Zárjuk külön ***Command*** leszármazott objektumba a kéréseket, és a menüelemeket ezzel paraméterezzük fel!

- Feladat (kitérő)
  - Tervezzük át a fenti osztályhierarchiát úgy, hogy alkalmas legyen almenü kezelésére!

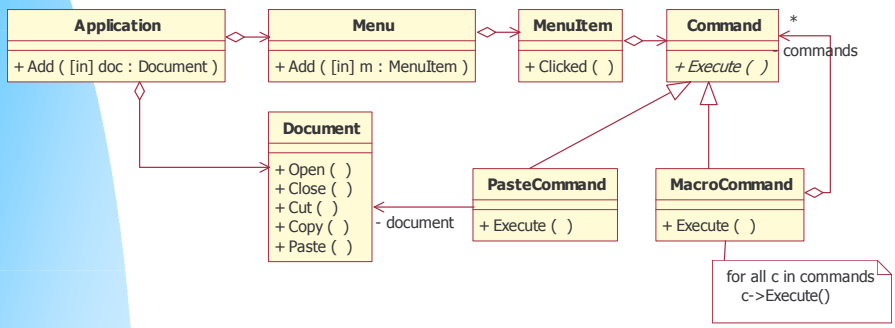
Command

- **Egy konkrét megvalósítás: Paste, Open**



Command

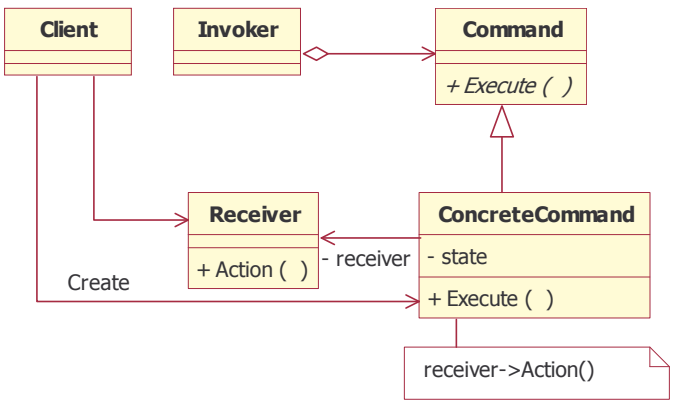
- **Macro: parancsok szekvenciája**



Command

- **Használjuk, ha**
  - Ha 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.
    - Specifikus eset adott szálból szeretnénk futtatni. Pl. .NET-ben GUI szálból, bár itt a Control.Invoke a megfelelő megoldás.
  - Visszavonás támogatására – eltároljuk az előző állapotot a command-ban

Command

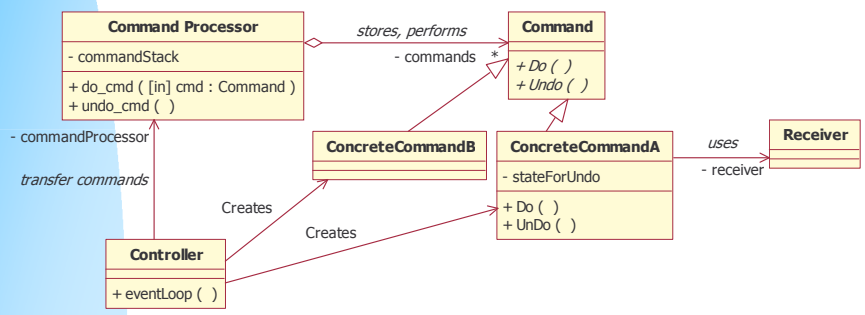


Command

- **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: tipikusan menüelem és toolbar gomb**
- **Könnyű hozzáadni új parancsokat, mert ehhez egyetlen létező osztályt sem kell változtatni. Hogyan tegyük ezt meg?**

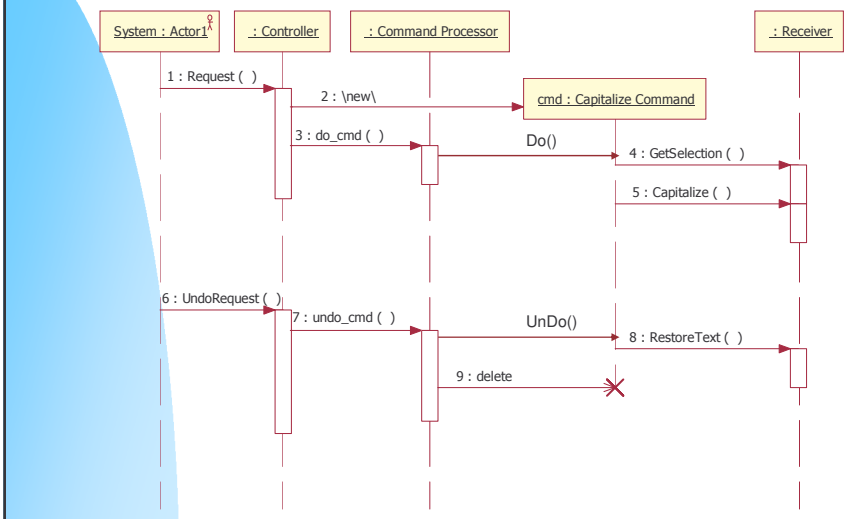
Command

- **Command Processor**
  - A Command egy változata
  - „Beépítve” támogatja az undo alapjait
  - Kulcsa a *Command Processor* osztály
    - Nyilvántartja a Command objektumokat
    - Aktiválja a Command objektumokat és egyéb szolgáltatásokat nyújt



Command

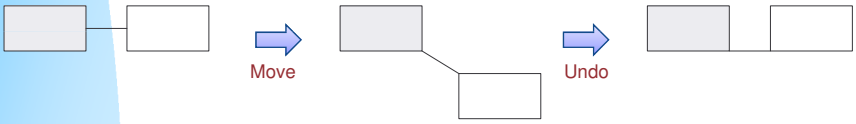
- **Command Processor**



Memento

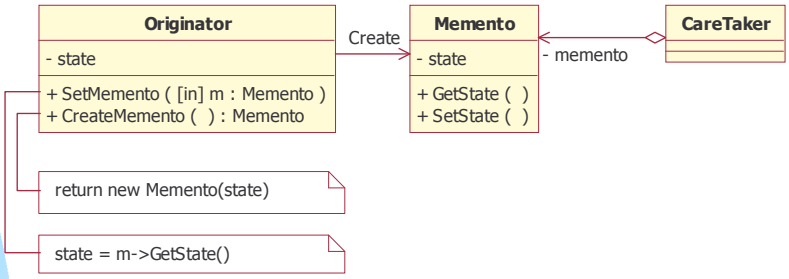
Memento

- **Cél**
  - Az egységbezárás megsértése nélkül a külvilág számára elérhetővé tenni az objektum belső állapotát. Így az objektum állapota később visszaállítható.
- **Példa: Visszavonás (undo) funkció a Dokumentumban**
- **Megoldás:**
  - Invertálható műveletek a Command mintában
    - Sokszor nehéz
    - Sokszor lehetetlen
      - Az objektum interfésze nem szolgáltat megfelelő publikus műveleteket



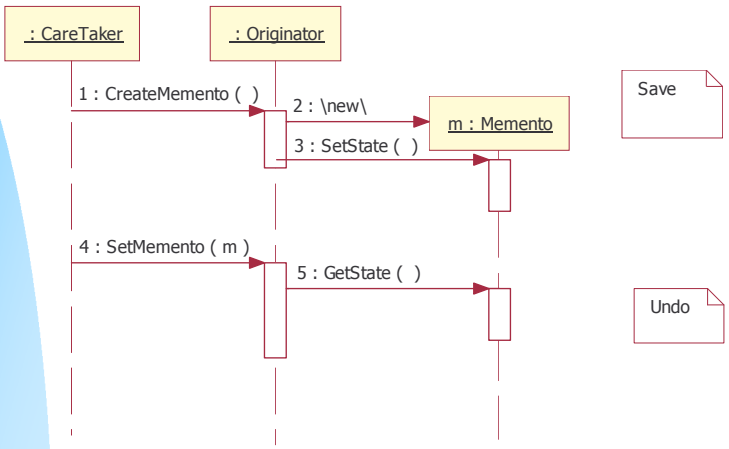
  - Probléma: nem az eredeti állapotot kaptuk vissza. Ez esetben csak egy megoldás van: meg kell jegyezzük a korábbi állapotot (a koordinátákat, az összekötő vonal kapcsolódási pontjait).

Memento



- ***Originator***: az ő állapotát kell tudni visszaállítani.
  - A ***CreateMemento***() elment (pontosabban visszaadja a state állapotot egy Memento objektum formájában)
  - A ***SetMemento***() visszaállít, (pontosabban beállítja a state állapotot a paraméterben megkapott Memento objektum alapján)
- ***Memento***: az Originator állapotát tárolja és elméletileg csak az Originator számára biztosít hozzáférést az állapothoz (state). Pl. C++-ban a friend alkalmazásával oldható ez meg.
- ***CareTaker***: nyilvántartja a Mementokat

Memento



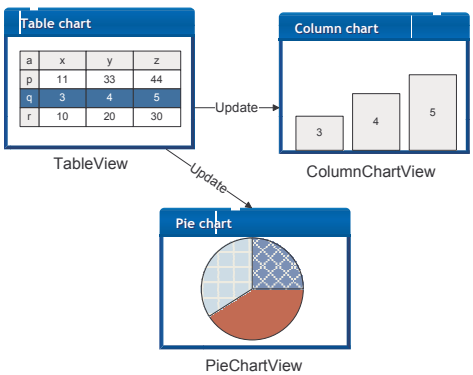
Memento

- **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
- **Előnyök:**
  - Megőrzi az egységbezárás határait
  - Egyszerűsíti az Originatort
- **Hátrányok:**
  - Memento használata sokszor erőforrásigényes
  - Nem mindig jósolható meg a Caretaker által lefoglalt hely, és ez sok is lehet

Observer

Observer

- **Cél**
  - Hogyan tudják objektumok értesíteni egymást állapotuk megváltozásáról anélkül, hogy függőség lenne a konkrét osztályaiktól
- **Példa: MVC vagy Document-View architektúra**
  - A felhasználó megváltoztatja az egyik nézeten az adatokat, hogyan frissítsük a többit? Közvetlen függvényhívással?



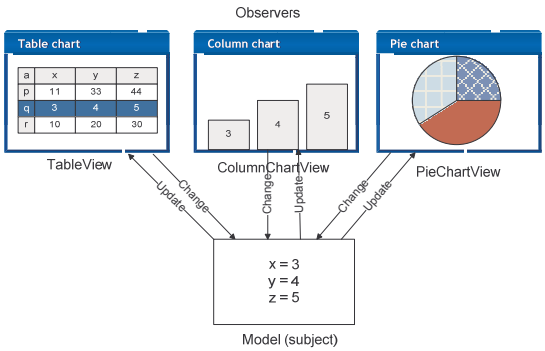
Observer

- **Közvetlen függvényhívás hátrányok**
  - Függőség a konkrét osztálytól. Pl. a ***TableView*** függ a ***ColumnChartView*** és a ***PieChartView*** osztályoktól
  - Ha új nézetet szeretnék bevezetni, minden nézet osztályt módosítani kell
  - A modell (üzleti logika) nem újrafelhasználható, mert össze van vonva a megjelenítéssel. Cél lenne, hogy úgy jelenjen meg, hogy ne legyen benne hivatkozás egy (konkrét) megjelenítési osztályra sem, mert akkor fel tudnánk több helyen használni
  - Nehéz karbantartani, továbbfejleszteni, újrafelhasználni, mert túl szoros a csatolás az osztályok között

Observer

Megoldás

- Az előző példára a MVC vagy Document-View architektúra
- Emeljük ki az adatokat és az azon értelmezett műveleteket egy modell osztályba
- A modellhez különböző view-kat (observers) lehet beregisztrálni
- Ha valamelyik view megváltoztatja a modell adatait, a modell értesíti az összes beregisztrált view-t a változásról.
- Az értesítés hatására a view lekérdezi a modell állapotát és frissíti magát
- A modell csak egy közös view (observer) interfészen keresztül tárolja a beregisztrált view-kat

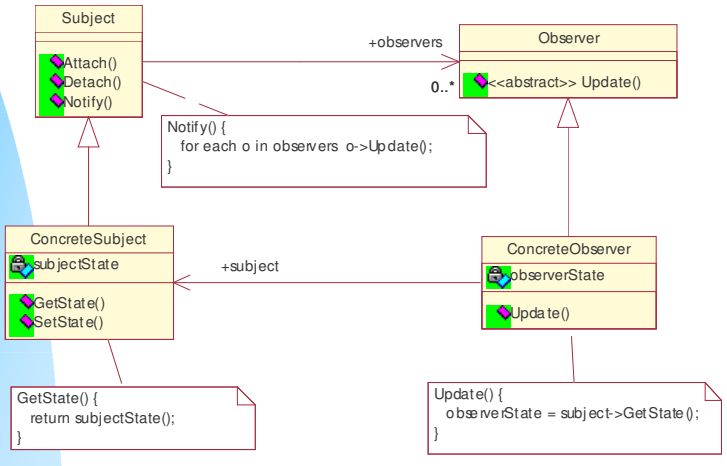


Observer

- **Előnyök**
  - A modell kódjában csak egy ***IView*** lista van, így a modell független az egyes ***IView-t*** implementáló osztályoktól. A modell újrafelhasználható!
  - Egy egyszerű mechanizmust kaptunk arra, hogy az összes view konzisztens nézetét mutassa az adatoknak.
  - A rendszer könnyen kiterjeszthető új view osztályokkal. Sem a modellt, sem a többi view osztályt nem kell ehhez módosítani.
- **Általánosítva, elvonatkoztatva a model-view esettől**
  - A fenti megközelítés lehetővé teszi, hogy egy objektum megváltozása esetén értesíteni tudjon tetszőleges más objektumokat anélkül, hogy bármit is tudna róluk
  - Ez az ún. *observer* minta lényege

Observer

Statikus nézet

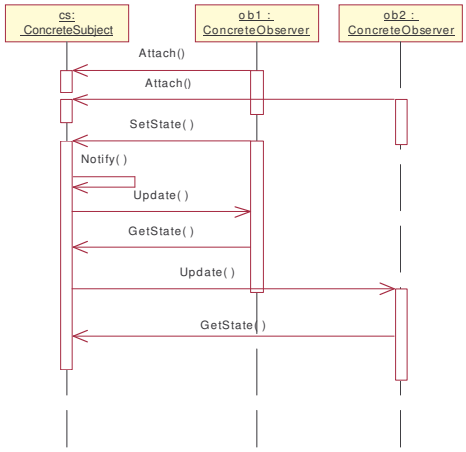


Observer

- **Szereplők**
  - Subject
    - Tárolja a beregisztrált Observer-eket
    - Interfészt definiál Observer-ek be- és kiregisztrálására valamint értesítésére
  - Observer
    - Interfészt definiál azon objektumok számára, amelyek értesülni szeretnének a Subject-ben bekövetkezett változásról (Update művelet)
  - ConcreteSubject
    - Az observer-ek számára érdekes állapotot tárol
    - Értesíti a beregisztrált observer-eket, amikor az állapota megváltozik
  - ConcreteObserver
    - Referenciát tárol a megfigyelt ConcreteSubject objektumra
    - Olyan állapotot tárol, amit a megfigyelt ConcreteSubject állapotával konzisztensen kell tartani
    - Implemetálja az Observer interfészét (Update művelet), ez az, amit a Subject meghív, amikor a ConcreteSubject állapota megváltozik. Ebben frissíti a saját állapotát a megfigyelt ConcreteSubject állapotának megfelelően

Observer

- **Dinamikus nézet – viselkedés (beregisztrálás és értesítés)**



- **„Sajnos” az UML szekvencia diagram csak objektumokat ábrázol, nem képes kifejezni, hogy milyen interfészen keresztül hivatkoznak egymásra az objektumok, így a függetlenséget nem tudja kifejezni**

Observer

- **Előnyök**
  - Laza kapcsolat a Subject és az Observer között
  - Üzenetszórás támogatása
- **Hátrányok**
  - Felesleges Update-ek
  - Az egyszerű Update alapján az Subject összes adatát le kell kérdezni (bár a Subject az Object::Update meghívásakor megadhatja paraméterként, hogy mi változott meg).
  - Az előbbi modellben a Subject ::Notify-t meghívó Observer-re is meghívódik az Update, holott az változtatta meg a ConcreteSubject állapotát

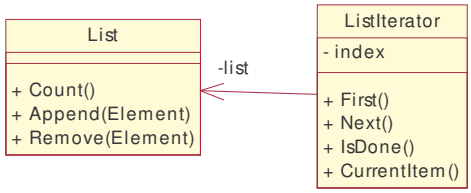
Observer

- **Használjuk, ha**
  - Amikor egy objektum megváltoztatása maga után vonja más objektumok megváltoztatását, és nem tudjuk, hogy hány objektumról van szó
  - Amikor egy objektumnak értesítenie kell más objektumokat az értesítendő objektum szerkezetére vonatkozó feltételezések nélkül
- **Megjegyzés: az Observer az egyik leggyakrabban használt minta**

Iterator

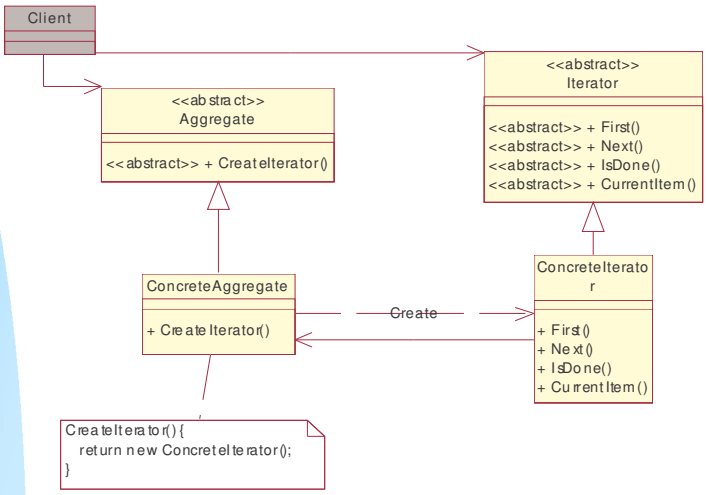
Iterator

- **Cél**
  - Szekvenciális hozzáférést biztosít egy összetett (pl. lista) objektum elemeihez anélkül, hogy annak belső reprezentációját felfedné
- **Példa: Lista**
  - Szeretnénk hozzáférni az elemeihez anélkül, hogy bármit is eláruljunk a belső struktúrától
  - Többféle módon is szeretnénk hozzáférni a listához
- **Megoldás**
  - Vegyük ki az elemeken való végiglépés műveleteit a lista interfészéből és helyezzük azokat egy másik, ún. Iterator objektumba



Iterator

- **Struktúra**



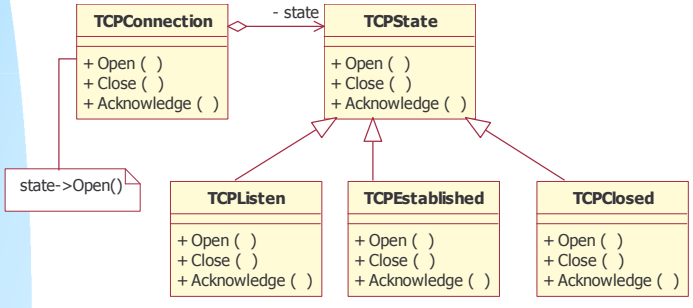
Iterator

- **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
  - Ha különböző tartalmazott struktúrákhoz szeretnénk hozzáférni hasonló interfésszel
- **Implementáció**
  - C++ -ban template osztállyal célszerű
  - Példa: az STL számos container és iterator osztályt szolgáltat

State

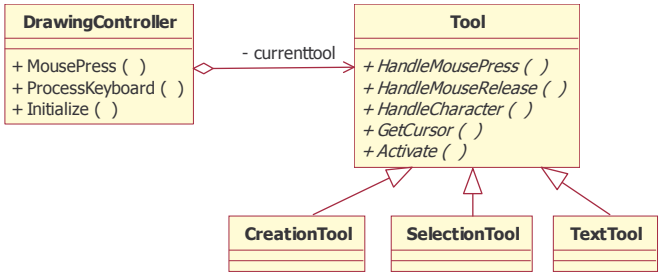
State

- **Cél**
  - Lehetővé teszi egy objektum viselkedésének megváltozását, amikor megváltozik az állapota.
- **Példa 1**
  - TCPConnection osztály egy hálózati kapcsolatot reprezentál
  - Három állapota lehet: Listening, Established, Closed
  - A kéréseket az állapotától függően kezeli
  - Megoldás:
    - if-ekkel
    - State pattern



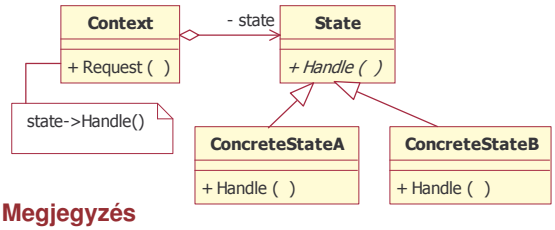
State

- **Példa 2**
  - Grafikus programban az eszköztár kezelése



State

- **Általános megoldás**



- **Megjegyzés**
  - Gyakori megoldás, hogy a Context átadja magát paraméterként a State műveleteinek

```
// C++ példa
void TCPConnection::Close ()
{
    state->Close(this);
}
```

  - A minta nem definiálja, ki felelős az állapotátmenetekért
    - Context
    - State leszármazottak. Ez elosztott megoldás. Jobb lehet, de ekkor kell ismerjék egymást a State leszámazott osztályok

State

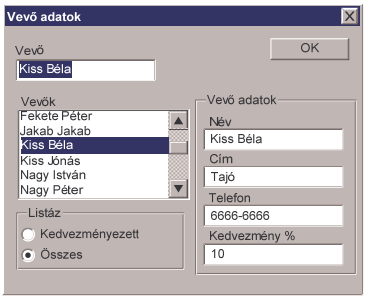
- **Használjuk ha**
  - Az objektum viselkedése függ az állapotától, és a viselkedését az aktuális állapotnak megfelelően futás közben meg kell változtatnia
  - A műveleteknek nagy feltételes ágai vannak, melyek az objektum állapotától függenek
- **Előnyök:**
  - Egységbezárja az állapotfüggő viselkedést, így könnyű új állapotok bevezetése
  - Áttekinthetőbb kód (nincs nagy switch-case szerkezet)
  - A State objektumokat meg lehet osztani
- **Hátrányok:**
  - Nő az osztályok száma (csak indokolt esetben használjuk)

Mediator

Mediator

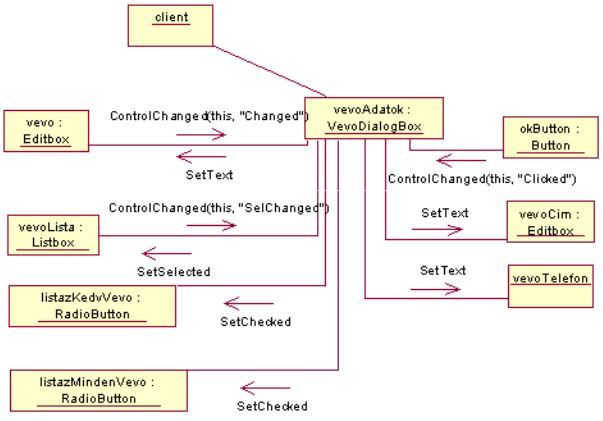
- **Cél**
  - Olyan objektumot definiál, ami egységbe zárja, hogy objektumok egy csoportja hogyan éri el egymást (hogyan kommunikál egymással). Megoldja, hogy az egymással kommunikáló objektumoknak ne kelljen egymásra hivatkozást tárolniuk, ezáltal biztosítja az objektumok laza csatolását.
- **Példa: Egy Form vagy dialógus ablak**

- Gyakran a formon levő vezérlőelem objektumok szoros kapcsolatban vannak egymással. Pl.
  - **a Vevő ablakba írás hatására a Vevők listboxban a megfelelő elem lesz a kiválasztott**
  - **A vevők listbox-ban egy elem kiválasztására a Vevő ablakba beíródik a megfelelő elem, valamint a „Vevő adatok" mezők is feltöltődnek a kiválasztott vevő adataival**
  - **A Listáz radiobutton-ok megfelelően szűrik a Vevők listbox tartalmát**



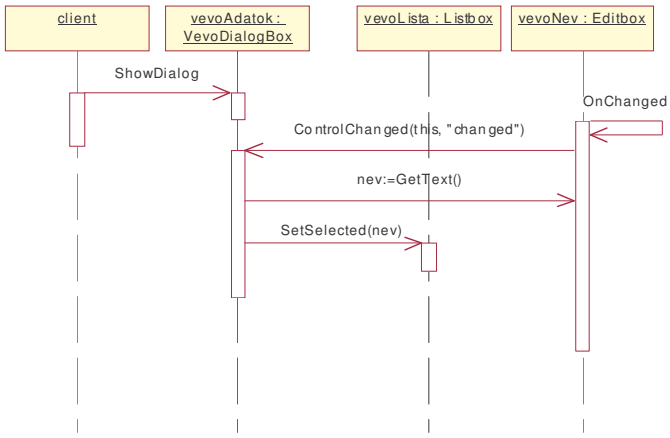
Mediator

- **Egy megoldás lehetne: minden objektum tartalmaz egy referenciát azokra, amelyek állapotát állítani szeretné**
  - A függőségi viszonyok „keszekuszák" lennének
  - Le kellene származtatni a Framework Edit, stb. osztályaiból, hogy a megfelelő referenciákat (vagy pointereket) hozzá tudjuk adni az egyes objektumokhoz: feleslegesen növelnénk az osztályok számát
- **Megoldás mediátor objektummal**



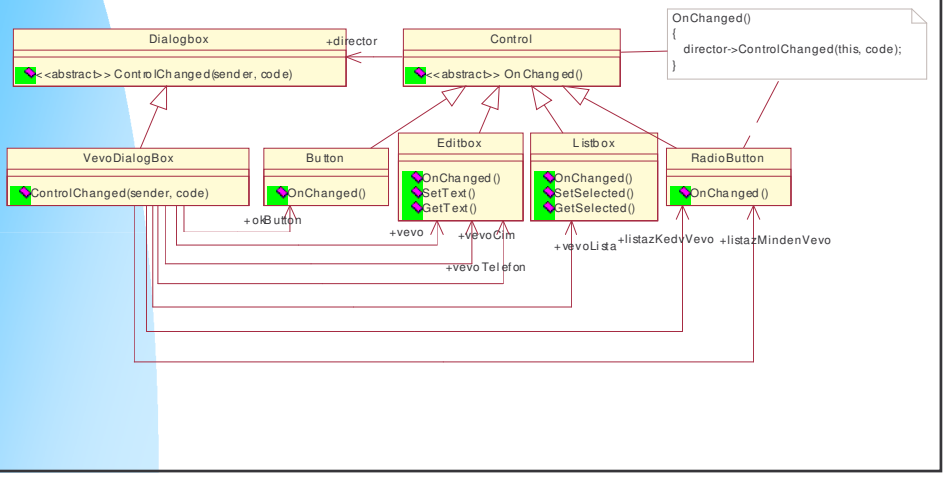
Mediator

- **Szerkezet**
  - Minden vezérlőelem tartalmaz egy referenciát a dialógus ablakra (mediátor)
  - A dialógus ablak minden vezérlőelemre tartalmaz egy referenciát
- **Dinamikus viselkedés**



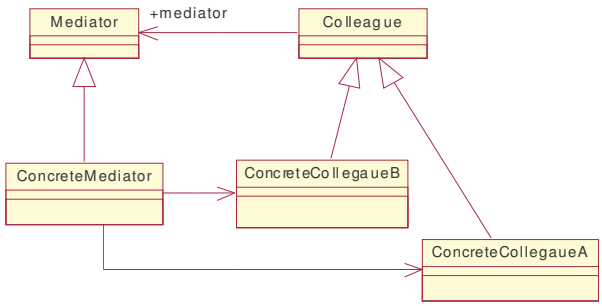
Mediator

- **Osztálydiagram**
  - Valamennyi osztály lehet a FrameWork része, csak a VevoDialogBox a fejlesztő által leszármaztatott osztály



Mediator

- **Osztálydiagram**
  - Valamennyi osztály lehet a FrameWork része, csak a VevoDialogBox a fejlesztő által leszármaztatott osztály
- **Az általános osztály diagram**



Mediator

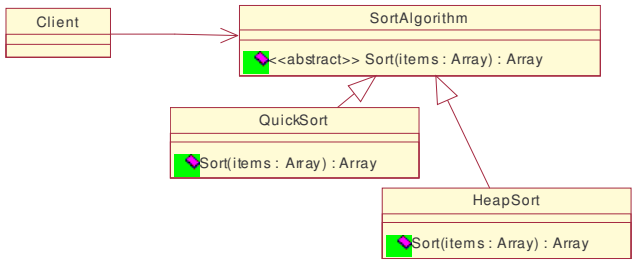
Alkalmazása

- **A Win32 dialógus ablakok**
  - A dialógus ablak egy mediator
  - A rajta levő vezérlőelemek a „colleague” objektumok
  - Ez egy nem OO alkalmazás
- **Visual Basic, Delphi, .NET Winform alkalmazások**
  - Delegátokon, illetve event-eken alapuló megoldás (ezek nyelvi szinten támogatottak)
  - A framework Form osztálya a mediator objektum, ebből kell az alkalmazásfejlesztőnek formonként egy saját osztályt leszármaztatnia. Ez a tagváltozóiban „tartalmazza” a rajta levő vezérlőelemeket
  - Minden vezérlőelem a típusától függő eventeket támogat (pl. az Edit az OnChanged-et, a Button az OnClicked-et), amire a Form osztályból leszármatatott osztályunk tagfüggvényeit tudjuk beregisztrálni.
- **Megjegyzés**
  - Probléma lehet: hajlamosak a programozók mindent a Form mediátor osztályba beleprogramozni
    - Ezek nagyon elhízott osztályok lesznek
    - Nem válik külön az alkalmazás logika és a megjelenítés

Strategy

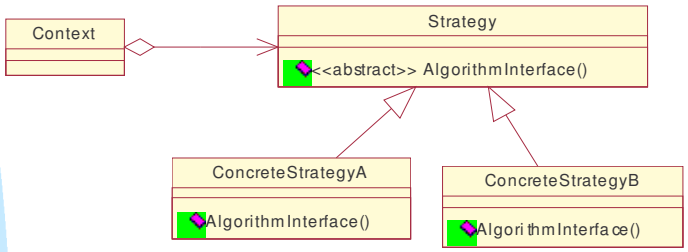
Strategy

- **Cél**
  - Algoritmusok egy csoportján belül az egyes algoritmusok egységbe zárása és egymással kicserélhetővé tétele. A kliens szemszögéből az általa használt algoritmusok szabadon kicserélhetőek.
- **Példa**
  - A kliens objektum különféle sorrendező algoritmusokat használhat
  - A kliens tartalmaz egy SortAlgorithm típusú pointert/referenciát egy konkrét (QuickSort v. HeapSort) objektumra.



Strategy

- Általános osztálydiagram



Pattern katalógus vége

Ami kimaradt

- **Creational**
  - Builder
- **Stuctural**
  - Flyweight
- **Behavioral**
  - Chain of Responsibility
  - Interpreter
  - Visitor

További tervezési minták

- **Elosztott, konkurens rendszerekre jellemző minták**
  - Szolgáltatás hozzáférés
  - Konfiguráció
  - Esemény kezelés
  - Szinkronizáció
  - Konkurencia
- **Valós idejű rendszerekre jellemző minták**

Összefoglalás

- **Tapasztalati tudást hordoznak**
  - Mi is rá tudunk jönni, de
    - Jó sokáig tart
    - Nem jövünk rá
    - Miért ne tanuljunk mások tapasztalataiból
  - Értékes tudás!
- **Célunk**
  - Ismerjünk meg minél több patternt
  - Hosszútávon legalább arra emlékezzünk, hogy egy adott problémakörben mely patterneket lehet jól használni