Szoftvertechnológia és -technikák

8. Előadás – Benedek Zoltán

Tervezési minták 3

BME AUT
Automatizálási és Alkalmazott Informatikai Tanszék

Tartalom

Tervezési minták

- Command
- Command Processor
- Memento
- Adapter
- Composite

Command (Parancs)

(Action)

Command

- Cél
  - Egy kérés **objektumként** való egységbezárása.
- Bővebben
  - Egy kérés általában egy **függvényhívásként** jelenik meg a kódban. Ezzel szemben Command esetében a kérés **objektumként** jelenik meg.
  - Ez lehetővé teszi a kliens különböző kérésekkel való felparaméterezését, a kérések sorba állítását, naplózását és visszavonását (undo).
- Alternatív név: Action
- Nagyon rendszerfüggő (C++, .NET UWP, Java, stb.) a koncepció értelmezése és az implementáció is

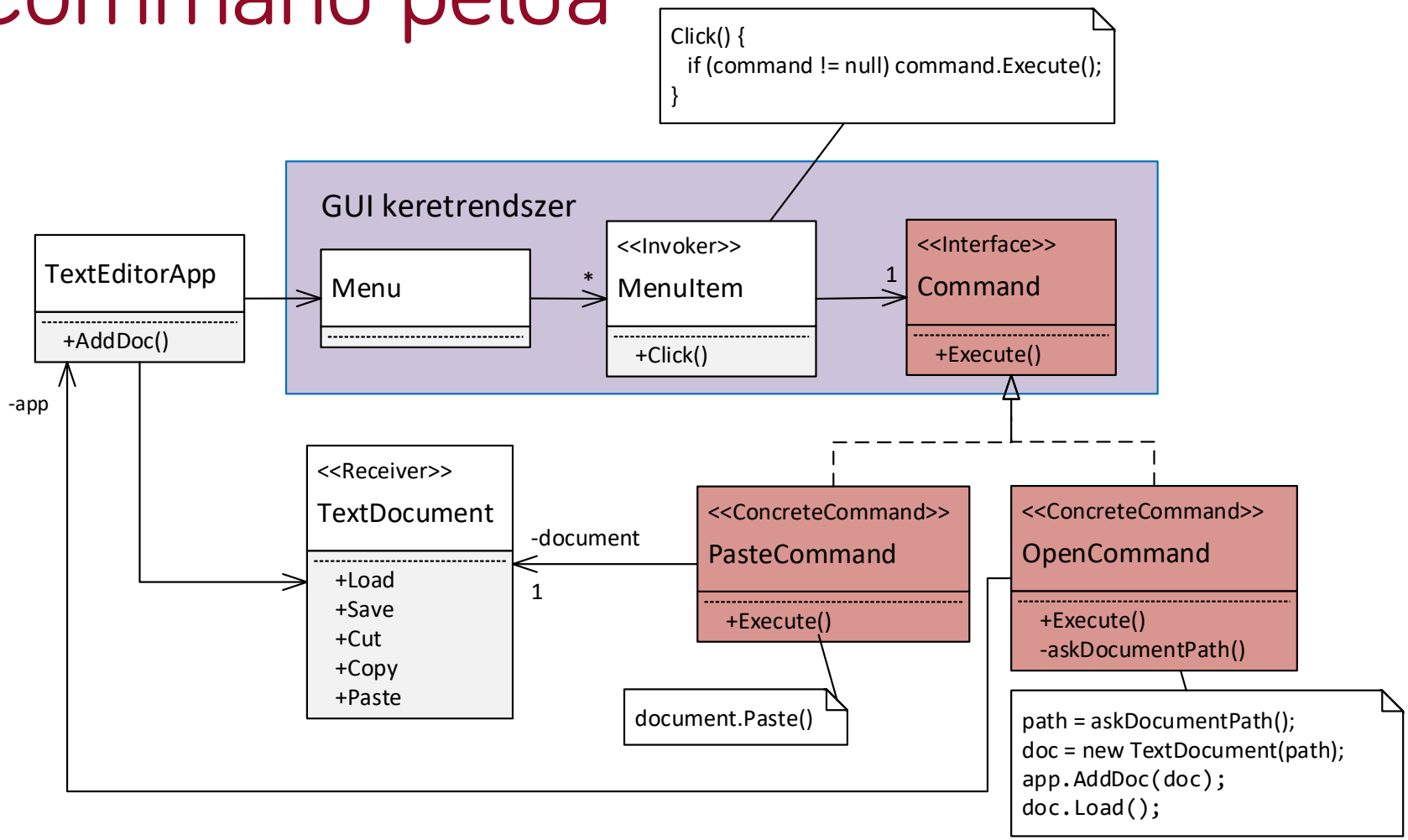
Példa

- 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?
    - Command minta, ezt nézzük most
    - Callback függvény – nem objektum-orientált (strukturált) megoldás
    - Delegate alapú megoldás - .NET
    - Adapter alapú megoldások – Java

Command Példa

- Adott egy GUI keretrendszer
  - Ebben beépített Menu (menü), MenuItem (menüelem) osztályok felhasználói parancsok futtatására
  - A feladat: az alkalmazásfejlesztő ugyanazon beépített MenuItem osztály objektumait teljesen eltérő , ráadásul alkalmazásfüggő kódok futtatására akarja használni. Pl.:
    - File/Open menüelem MenuItem objektuma a fájlmegnyitás kódját kell futtassa
    - Edit/Paste menüelem MenuItem objektuma a paste (beillesztés) logikához tartozó kódot kell futtassa
  - Ha ugyanaz az osztály (MenuItem)-> ugyanazok a tagfüggvények. **Hogyan lehet mégis eltérő kódot futtatni eltérő MenuItem objektumok esetén**?

Command példa



Zárjuk külön Command interfészt implementáló osztálybeli objektumokba a kéréseket, és a menüelemeket ezekkel paraméterezzük fel

Magyarázat

Command példa magyarázat

- A példa azt illusztrálja, hogy a Command minta használatával hogyan lehet a File/Open és az Edit/Paste menüelemekhez a megfelelő kód futtatását elérni egy szövegszerkesztő alkalmazásban.
- `TextEditorApp` osztály: magát az alkalmazást reprezentálja. Tartalmaz egy Menu objektumot, ez az alkalmazás menüje.
- A Menu több menüelem (`MenuItem`) objektumból áll (pl. File/Open, Edit/Paste).
- A GUI keretrendszerben bevezetünk egy `Command` interfészt egyetlen, `Execute` metódussal
- Az általános `MenuItem` osztálynak van egy hivatkozása egy `Command` objektumra. Ez a kezdetben null, de bármilyen `Command` interfészt implementáló osztálybeli objektumra ráállítható.

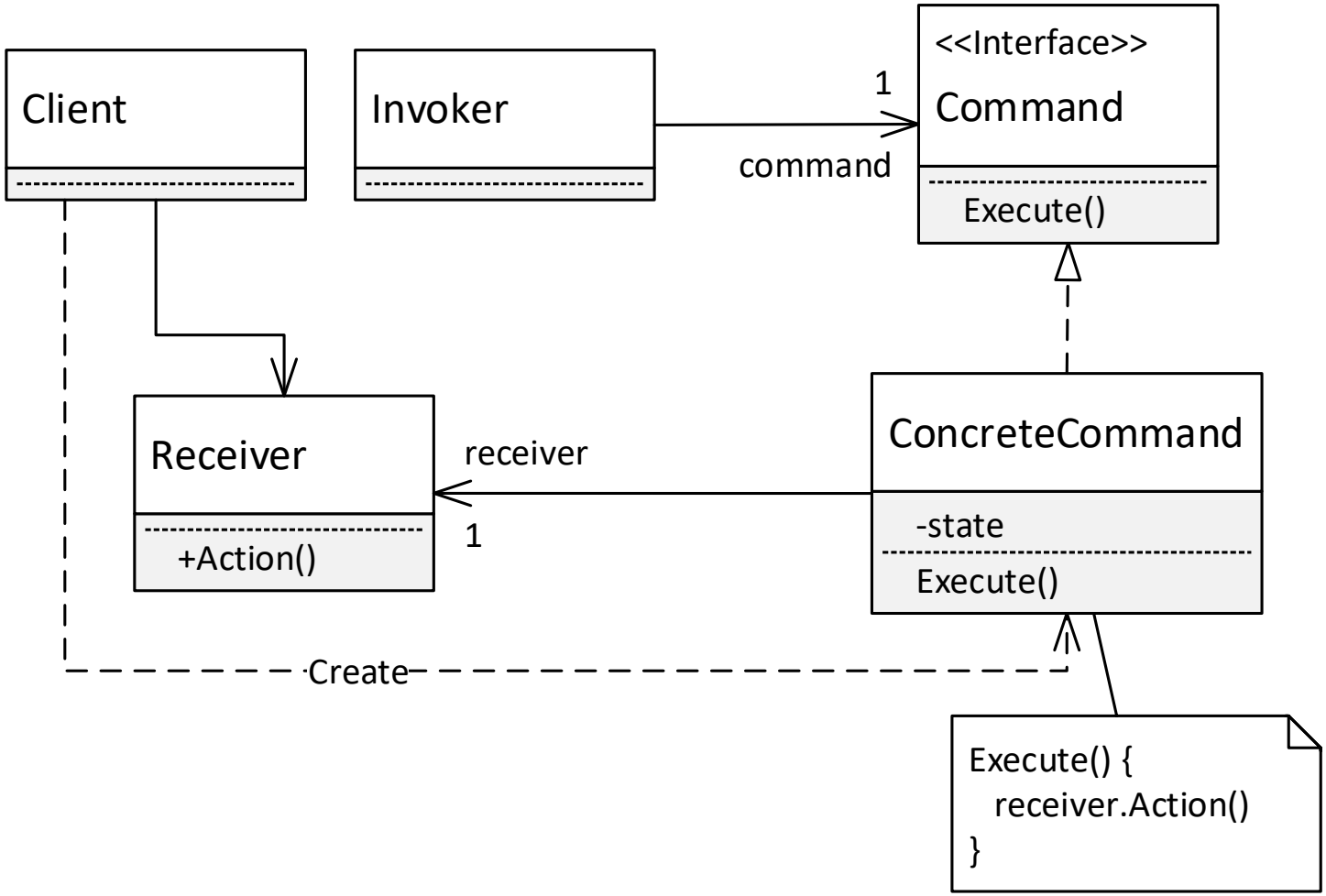
Command példa magyarázat

- Minden „parancshoz” bevezetünk egy `Command` implementációt: a Paste-hez egy `PasteCommand` osztályt, az Open-hez egy `OpenCommand` osztályt. Ezek Execute műveletébe a parancsspecifikus kódot tesszük (`PasteCommand.Execute`–> beillesztés megvalósítása, `OpenCommand.Execute` -> fájlmegnyitás megvalósítása).
- A Paste `MenuItem` command hivatkozását egy `PasteCommand` objektumra állítjuk, az Open `MenuItem`-ét egy `OpenCommand`-ra
- Amikor a felhasználó kattint egy adott menüelemen, meghívódik a `MenuItem` `Click` művelete, mely meghívja a `MenuItem` objektumhoz beregisztrált command objektum `Execute` műveletét.
- Ezzel pont elértük a célunkat: futás közben a Paste `MenuItem` kattintás a hozzá beregisztrált `PasteCommand.Execute` meghívását eredményezi -> ez beilleszti a szöveget. Ezzel analóg módon az Open `MenuItem` kattintás a hozzá beregisztrált `OpenCommand.Execute` meghívását eredményezi -> ez megnyit egy új dokumentumot.
- Általánosságában: **Azáltal, hogy ugyanolyan `MenuItem` osztálybeli objektumokhoz eltérő `Command` implementációkat regisztráltunk be, eltérő `Execute` művelet fut le, vagyis el tudtuk érni, hogy a kattintás eseményre más-más kód fusson le.**
- Megjegyzés: a kapcsolódó C# mintakódban a `Command` interfész neve `ICommand` követve a .NET konvenciókat.

Command megjegyzések

- Példakód: lásd DesPattCode/Command mappa (a futtatáshoz nevezzük át a Program osztály Main2 függvényét Main-re).
- Ízlés kérdése , mennyi logikát teszünk a `Command.Execute`-ba. Két megközelítés lehetséges
  - Beletesszük a logika részletes implementációját. Erre példa az `OpenCommand.Execute`.
  - A részletes implementációt más osztályba (ún. „`Receiver`”) tesszük, a `Command.Execute` ennek delegálja a kérést. Erre példa a `PasteCommand.Execute`, mely a kérdéseket a `TextDocument` osztálynak továbbítja.
- A Menu és MenuItem nem szükséges része a Command mintának, a minta szempontjából lényegtelen, mi futtatja a parancsot.

Command minta általánosságában



Command – mikor használjuk

- Használjuk, ha
  - Ha strukturált programban callback függvényt használnánk, objektumorientált programban használjunk commandot helyette. Más megközelítésben: ezzel tudtunk objektumspecifikus – és nem osztályspecifikus - kódot futtatni (ilyen volt a MenuItem példa is).
  - **Visszavonás támogatására** – eltároljuk az előző állapotot a command-ban. Lásd Command Processor minta rövidesen!
  - **Szeretnénk a kéréseket különböző időben kiszolgálni (a parancs kiadásától, megszületésétől „leválasztva”)**. Ilyenkor várakozási sort használunk, ebbe tesszük a command objektumokat. Az egyes command objektumokban tároljuk a parancs paramétereket, majd akár különböző folyamatokból/szálakból is futtathatjuk őket.

Command további gondolatok

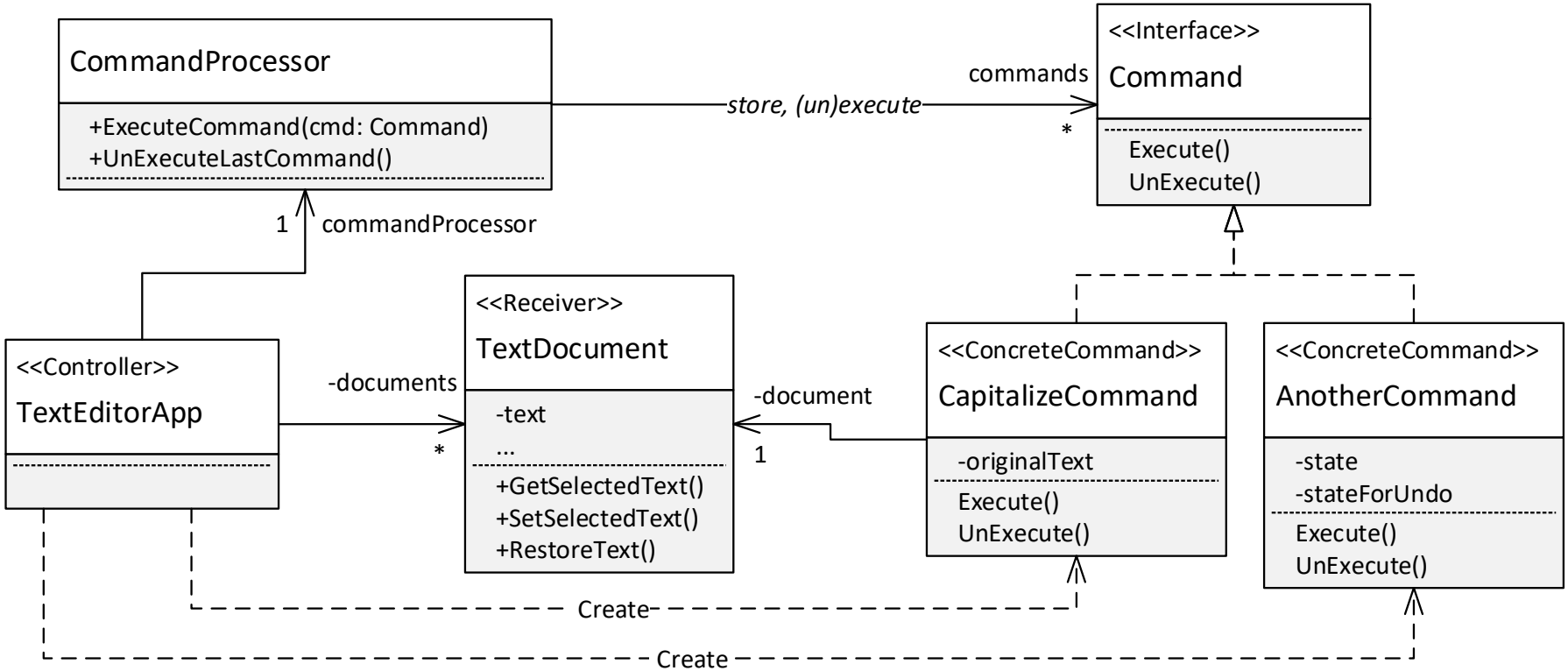
- Általánosítva az alapgondolata a következő: Elválasztja a parancsot **kiadó** objektumot (pl. MenuItem) attól az objektumtól, amelyik tudja, hogyan kell **lekezelni** (adott Command implementáció).
- Könnyű hozzáadni új parancsokat, mert ehhez egyetlen létező osztályt sem kell változtatni. Hogyan tegyük ezt meg?
- Összetett parancsok támogatása (lásd Composite minta később)
- Ismétlésképpen: nagyon rendszerfüggő (.NET UWP, Java, stb.) a koncepció értelmezése, sokszor némiképpen mást értenek alatta!
  - Pl. UWP-ben is mást jelent, nincs minden parancsfuttatáshoz külön Command objektum létrehozva, viszont a parancs futtatásán túl kezelni, hogy az adott parancs az adott pillanatban engedélyezve vagy tiltva van-e.

Command Processor (Parancsfuttató)

Command Processor

- A Command minta egy változata
- „Beépítve” támogatja parancsok visszavonását (Undo)
- Alapelvek
  - UnExecute művelet bevezetése a Command interfészbe
    - Minden command objektumnak támogatnia kell a változtatásának visszavonását is az UnExecute művelethez
    - Minden command objektum az Execute során eltárolja tagváltozókban azt az állapotot, mely a visszavonáshoz szükséges
  - CommandProcessor osztály bevezetése
    - Eltárolja a már futtatott command objektumokat, hogy ha később a parancs visszavonására kerül sor, rendelkezésre álljon a command objektum
    - A parancsok futtatását és visszavonását rajta keresztül végezzük (ExecuteCommand és UnExecuteLastCommand műveletek)

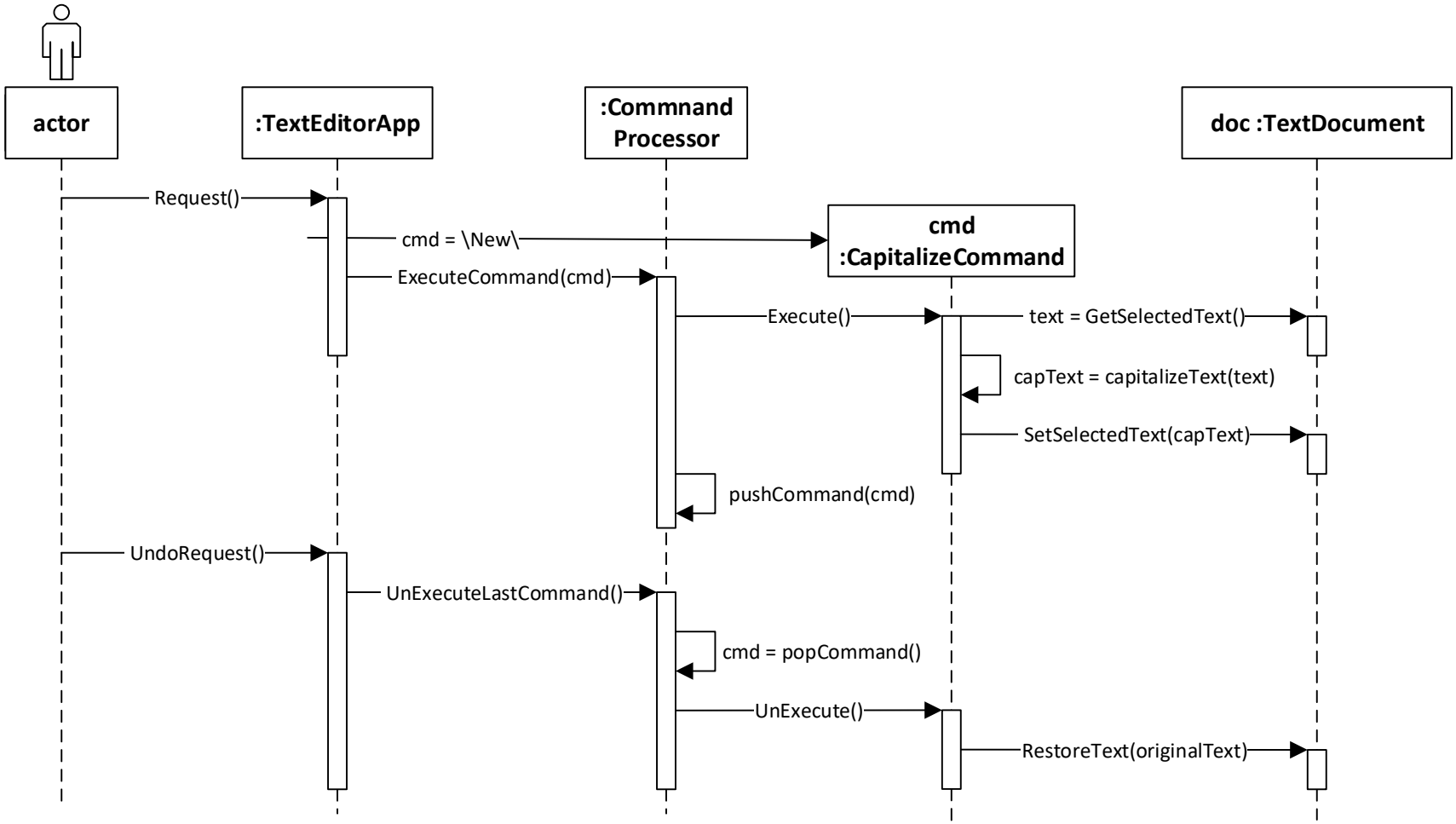
Command Processor példa



Capitalize parancs: a kijelölt szöveget nagybetűssé alakítja

Command Processor példa

- Kijelölt szöveg nagybetűssé alakításának lépései majd visszavonása



Példa magyarázat

- Kijelölt szöveg nagybetűssé alakításának lépései
  1. A felhasználó kijelöl egy szöveget és kéri ennek nagybetűssé alakítását. Lényegtelen, milyen módon, a példánkban a `TextEditorApp` fogadja a kérést.
  2. A TextEditorApp létrehoz egy `CapitalizeCommand` objektumot, és meghívja a **`CommandProcessor.ExecuteCommand`** műveletét, paraméterként átadva neki a parancs objektumot. Ez a művelet:
     1. Meghívja a command objektum `Execute` műveletét (így lefut a parancs kódja – a példában nagybetűssé alakítja a kijelölt szöveget)
     2. Eltárolja a parancs objektumot egy command gyűjteményben a `pushCommand` művelettel (hogy az esetleges későbbi Undo során meglegyen)

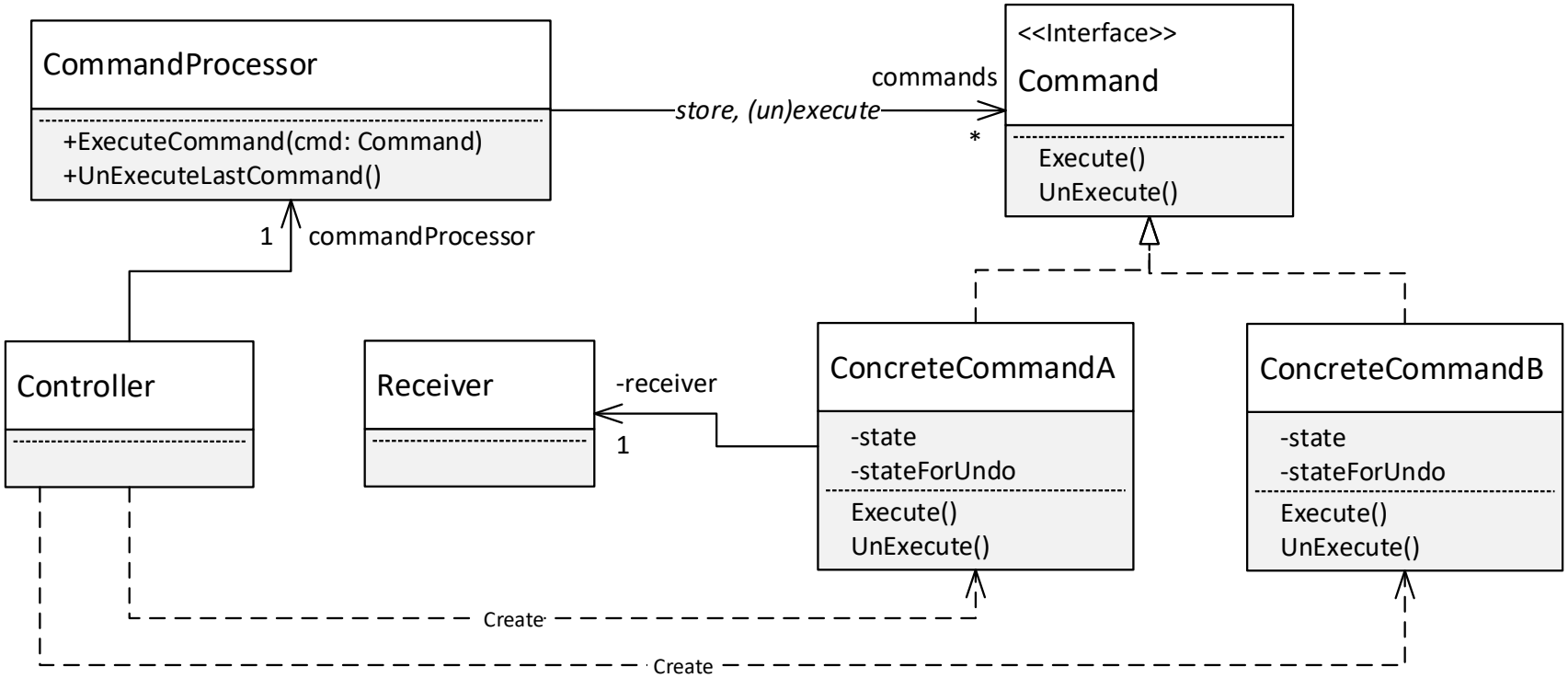
Példa magyarázat

- A felhasználói visszavonás (Undo) lépései
  1. A felhasználó kéri az utolsó parancs visszavonását. Lényegtelen, milyen módon, a példánkban a `TextEditorApp` fogadja a kérést.
  2. A `TextEditorApp` meghívja a **`CommandProcessor.UnExecuteLastCommand`** műveletét. Ez a művelet:
     1. Kiveszi a legutoljára eltárolt parancs objektumot a parancs gyűjteményéből (`popCommand` művelet)
     2. Erre a parancs objektumra meghívja az `UnExecute` műveletet (így lefut a parancs azon kódja, mely visszacsinálja a parancs által korábban végrehajtott változtatásokat)

Példa

- Kód: lásd DesPattCode/CommandProcessor mappa
  - A kód futtatható és debugolható, a Program osztályban a Main2 függvényt nevezzük át Main-re

Command Processor általánosságában



Command Processor megjegyzések

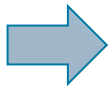
- A Command két műveletét Execute-nak és UnExecute-nak neveztük. Szokásos a Do és Undo nevek használata is, illetve az UnExecute-ra a Revert alternatíva.
- A parancs végrehajtás során törekedjünk arra, hogy a visszaállításhoz csak azon állapotot mentsük el, mely a visszaállításhoz mindenképpen szükséges
  - Ha pl. egy szövegszerkesztő esetén minden parancs során a teljes dokumentum tartalmát elmentjük, akkor nagyméretű dokumentum esetén hamar kifuthatunk a memóriából (vagy csak nagyon kevés lépés visszavonását tudjuk támogatni).

Memento

Memento

- Cél
  - Az egységbezárás megsértése nélkül a külvilág számára elérhetővé tenni az objektum belső állapotát:
    - Vagyis pl. anélkül, hogy a védett változókat publikussá tennénk
    - Így az objektum állapota később visszaállítható
- Példa: Visszavonás (Undo) funkció megvalósítása egy dokumentum esetén

Célok kifejtése



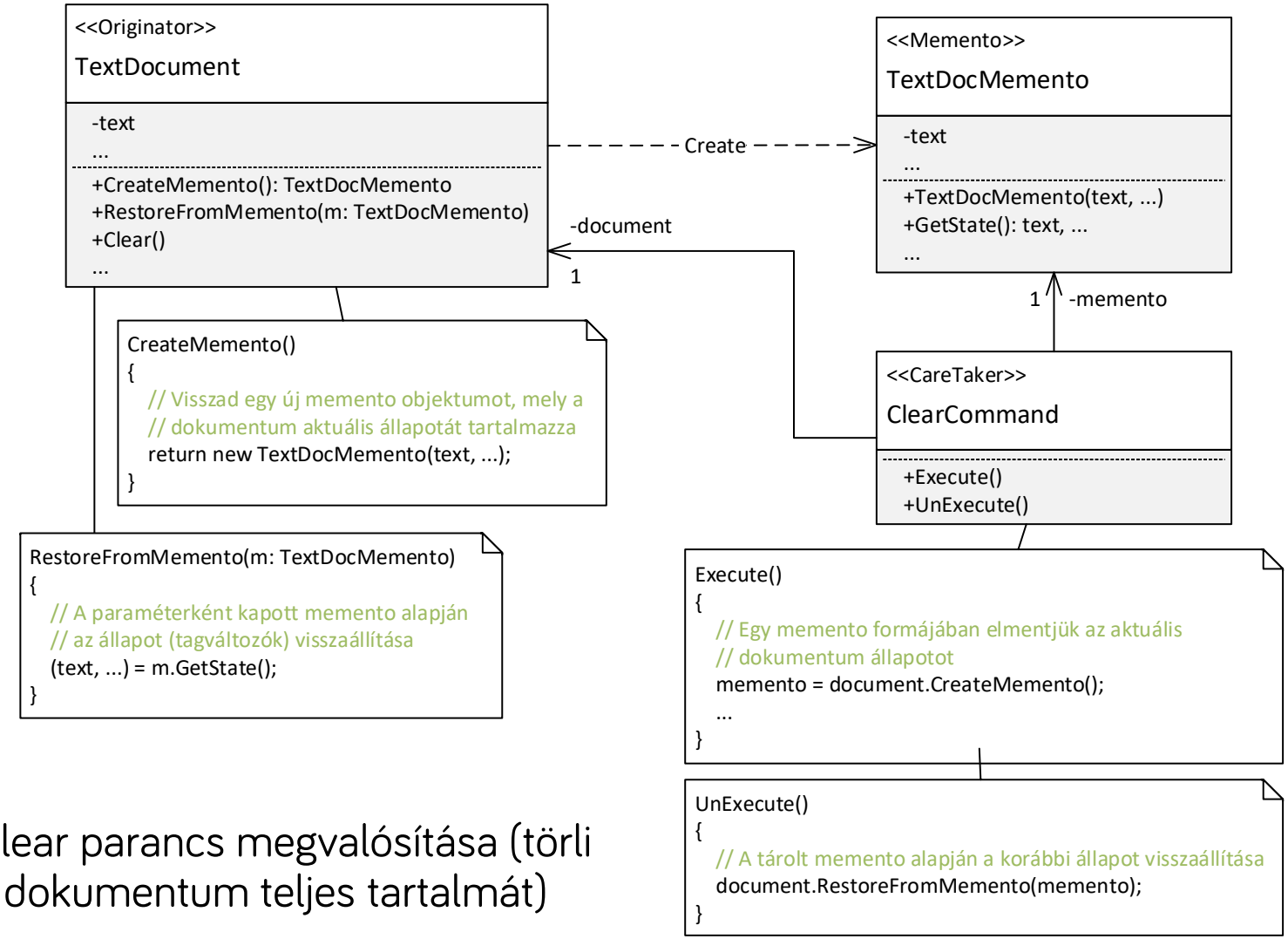
Memento

- Visszavonható műveletek
- Bár a Memento a Command/Command Processor minta nélkül is használható, jellemzően ezekkel célszerű kombinálni, mi is ezt tesszük
  - A visszavonás sokszor nehéz vagy lehetetlen anélkül, hogy az objektum (pl. dokumentum) **teljes állapotát** elmentenék, majd visszaállítanánk a visszavonás során (pl. dokumentum teljes tartalmát törlő „Clear” parancs visszavonása).
  - Az objektum teljes állapota viszont általában nem elérhető más osztályok számára, mert az egységbezárás miatt a tagváltozók védettek (private).
  - Csak az visszavonáshoz való állapotmentés lehetősége miatt kellene ezeket a változókat publikussá tenni. **Nem tesszük (teljesen szembe menne az egységbezárás elvével)!** Inkább alkalmazzuk a Memento mintát.
- A Memento minta lényege, hogy egy objektum (pl. dokumentum) adott állapotát egy Memento objektumba csomagoljuk be, és ilyen „becsomagolt” formában tesszük elérhetővé (a visszavonás megvalósításához)

Memento példa

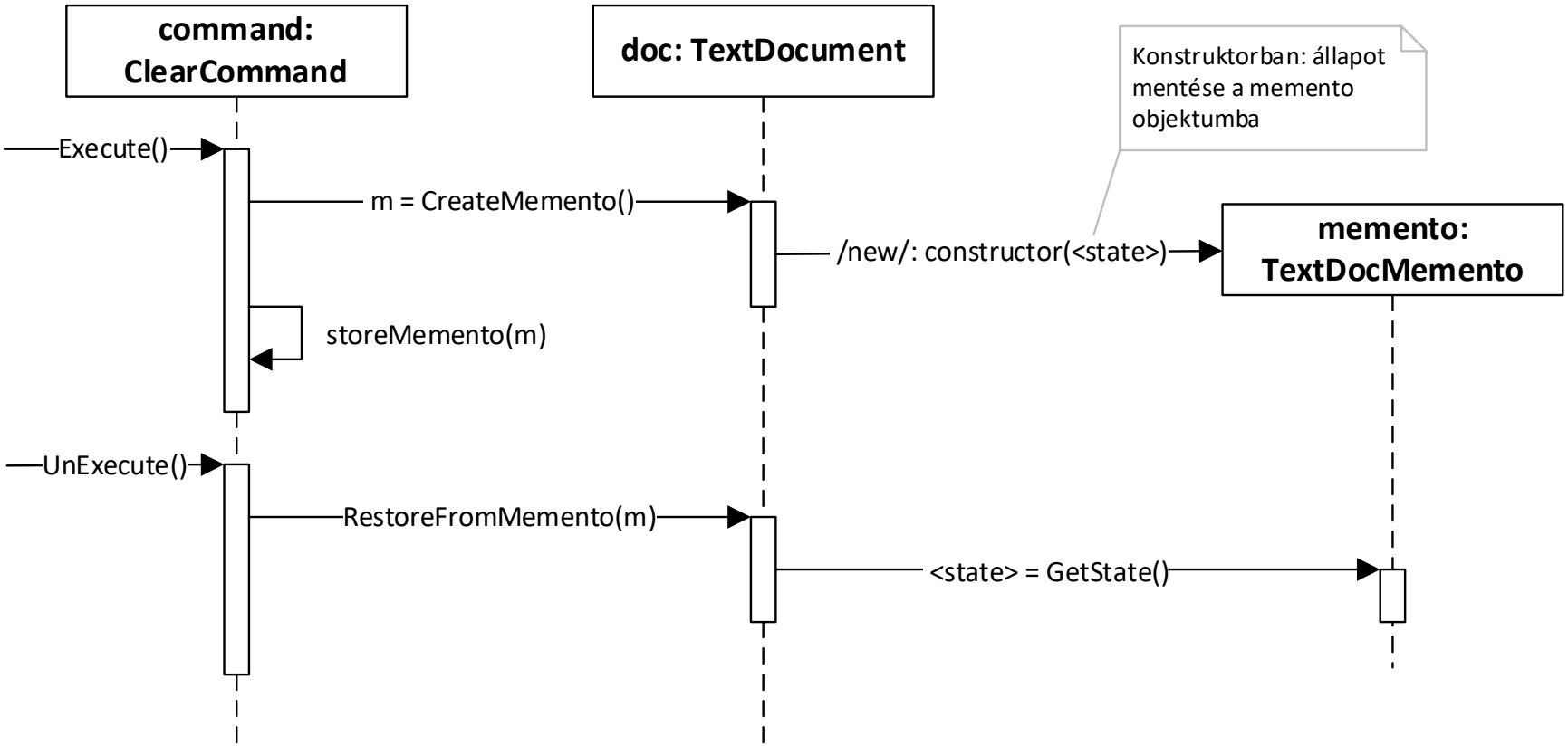
- Feladat : Clear parancs megvalósítása (törli a dokumentum teljes tartalmát)
- Ehhez bevezetjük a ClearCommand parancs osztályt (Command Processor minta, `Command` implementáció)
- A `ClearCommand` az `Execute` műveletében el kell mentse a TextDocument teljes állapotát, és `UnExecute` során vissza is kell állítsa
  - De a `TextDocument` nem fér hozzá a `TextEditor` állapotához (text és egyéb tagváltozók private-ok és nincs mindenhez lekérdező/beállító függvény sem)
  - Nem is akarjuk ezeket publikussá/közvetlen hozzáférhetővé tenni (egységbezárás megőrzése)
  - Helyette: bevezetünk egy memento (`TextDocMemento`) osztályt. Ennek minden objektuma a dokumentum adott időpontbeli állapotát tárolja a tagváltozóiban (a tagváltozói „tükörképei” a dokumentum tagváltozóinak)

Memento példa



Clear parancs megvalósítása (törli a dokumentum teljes tartalmát)

Memento szekvenciadiagram



Megjegyzés: a <state> az ábrán a mentendő dokumentum állapotot jelenti (text és egyéb tagváltozók, melyeket a clear parancs módosít)

Memento példa

- Parancs (Clear példa) futtatása lépések (a `ClearCommand.Execute` a „belépési pont”)
  1. Egy memento formájában elmentjük az aktuális dokumentum állapotot
     - `document.CreateMemento()` hívása, mely visszaad egy új memento objektumot, mely a dokumentum aktuális állapotát tartalmazza a tagváltozóiban
     - A parancs objektum elmenti egy tagváltozóba ezt a memento objektumot a későbbi visszaállításhoz
  2. Lefut a parancs lényegi kódja, a clear „kipucolja” a dokumentum belső állapotát (text és egyéb tagváltozók kezdőértékre állítása)

Memento példa

- Parancs (Clear példa) visszavonása lépések (a `ClearCommand.UnExecute` a „belépési pont”)

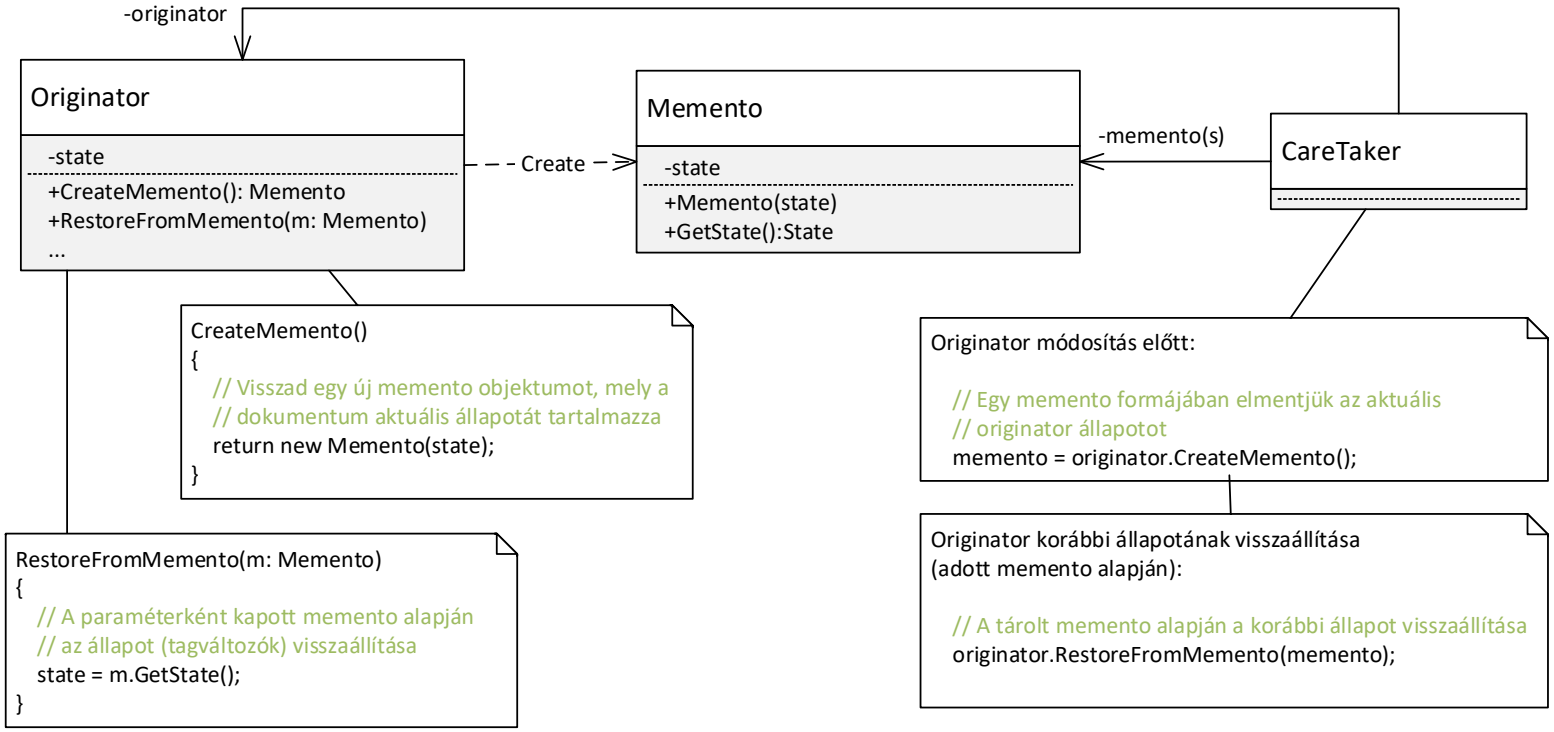
  Visszaállítjuk a korábbi dokumentumállapotot

  - A tagváltozóban tárolt memento objektumot paraméterként átadva `document.RestoreFromMemento()` hívása
  - A `document.RestoreFromMemento()` a paraméterként kapott memento alapján visszaállítja a dokumentum korábbi állapotát (tagváltozók értékének visszaállítása)

Memento példa

- Kód: lásd DesPattCode/ MementoWithCommandProcessor mappa (a futtatáshoz nevezzük át a Program osztály Main2 függvényét Main-re).
- Ez kicsit összetettebb implementáció, mint amit az előző diákon néztünk
  - A `TextDocument` osztálynak nem csak a text tagváltozó adja az állapotát, hanem a `selectionStartIndex` és `selectionLenght` tagok is (ezek határozzák meg az aktuális kijelölést a dokumentumban)
  - Ezeket is elmentjük a memento osztályunkba és visszaállítjuk az `UnExecute` során
  - Így kicsit életszerűbb a példa
  - A példa futtatható, az App/Program osztály Main2 függvényét kell Main-re nevezni a futtatáshoz és debuggoláshoz

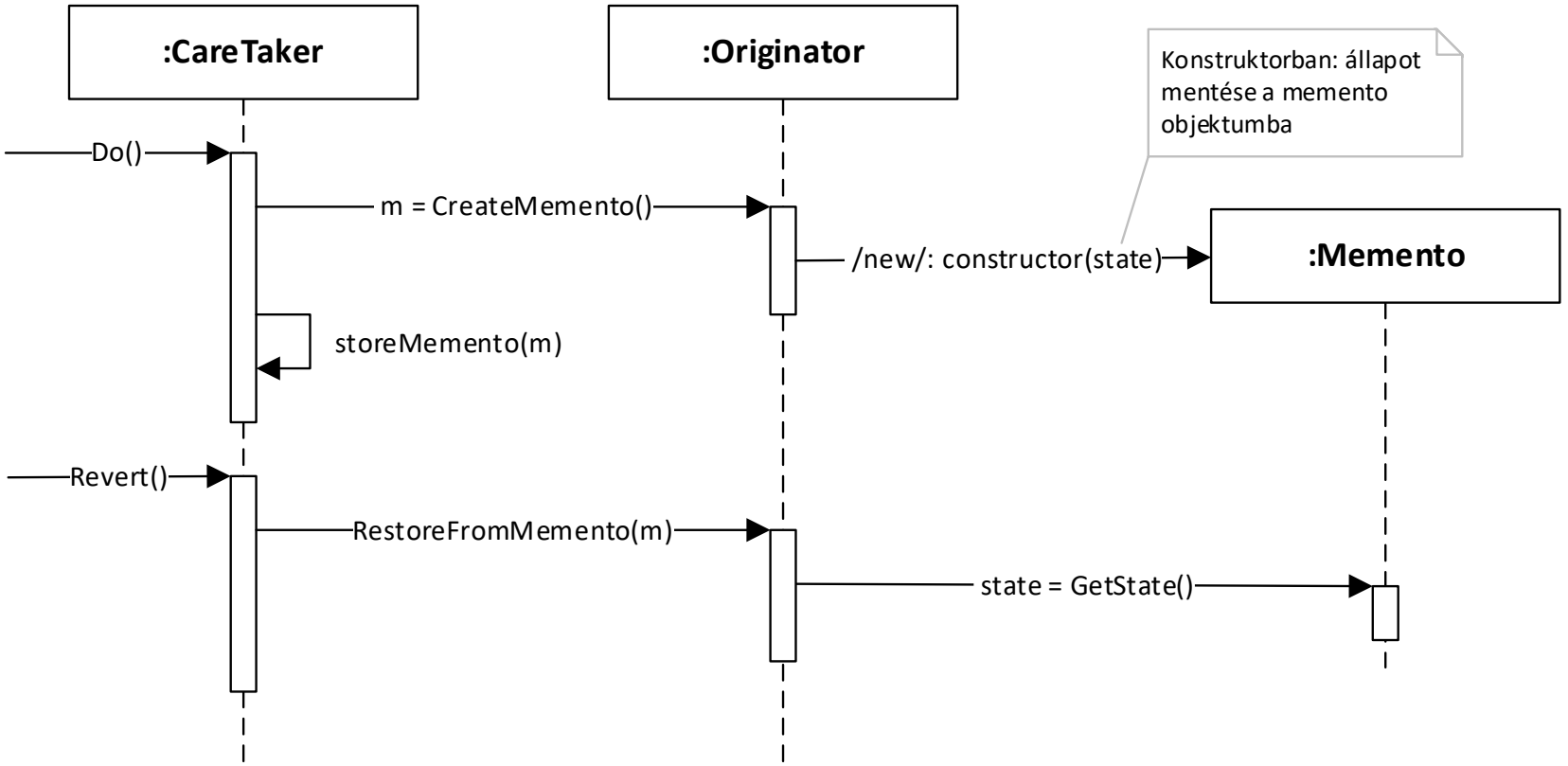
Memento minta általánosságában



- 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), de e nem minden programozási nyelven megvalósítható.
- CareTaker: nyilvántartja a mementot/mementokat (nem kell feltétlen command legyen)

Memento minta általánosságában

- Szekvenciadiagram



Memento megjegyzés

- Eddig azt mondtuk, hogy a Memento abban segít, hogy ne kelljen publikussá tenni az Originator (pl. TextDocument) tagváltozóit
  - Nem tesszük publikussá
  - És nem is vezetünk be olyan publikus műveleteket, melyekkel direktben le lehetne kérdezni és be lehetne állítani őket (ez is sértené az egységbezárást)
    - Vagyis marad a Memento, mint célszerű megoldás

Memento

- Használjuk, ha
  - Egy objektum (rész)állapotát később vissza kell állítani és ennek támogatásához meg kellene sérteni az objektum egységbezárását

- Előnyök:
  - Megőrzi az egységbezárás határait

- Hátrányok:
  - Memento használata sokszor erőforrásigényes (pl. teljes dokumentum állapot mentése sok példányban)

Adapter (Illesztő)

(Wrapper)

Adapter

- Cél
  - Egy osztály interfészét olyan interfésszé konvertálja, amilyent a kliens vár. Lehetővé teszi olyan osztályok együttműködését, melyek egyébként az inkompatibilis interfészeik miatt nem tudnának együttműködni.
- Alternatív név: Wrapper
- Elve

?

Nem megfelelő interfész

Megoldás: adapter

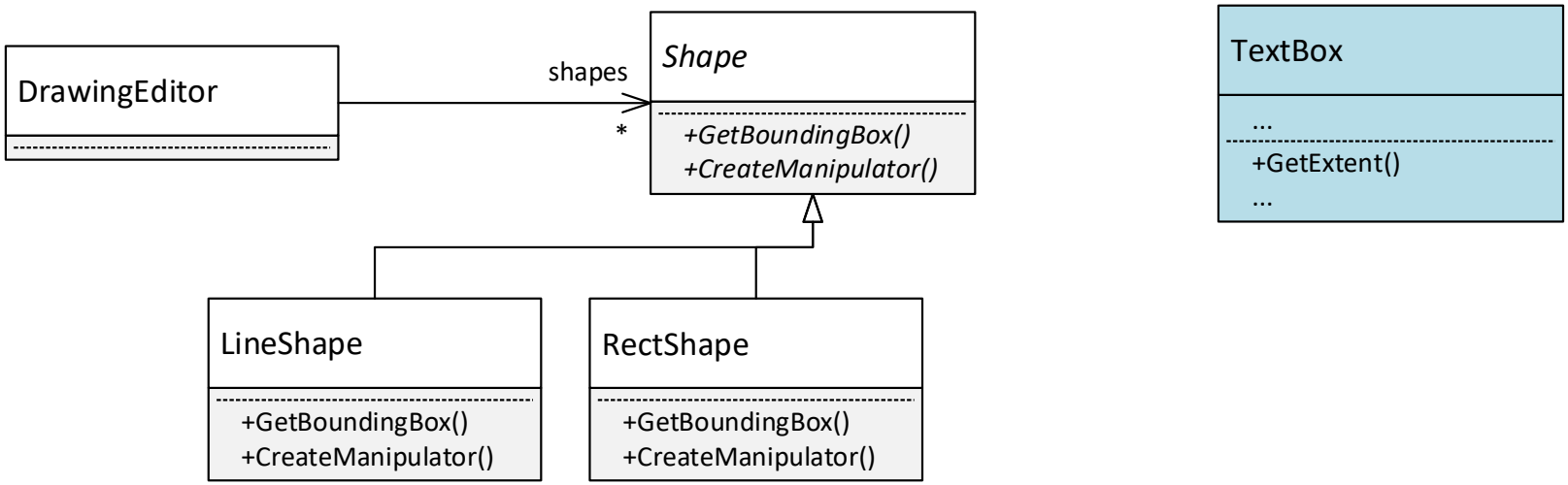
Adapter - példa

- A feladatunk egy vektorgrafikus alkalmazás elkészítése
  - A felhasználó különböző grafikus alakzatokat tud elhelyezni a felületen. Támogatni kell a vonal, téglalap, stb. alakzatot, valamint a **szerkeszthető szövegdobozt** is (amibe lehet gépelni futás közben)
  - Bevezetünk egy **Shape ősosztályt**, hogy az **alakzatokat egységesen szeretnék kezelni (**esetünkben **egy heterogén kollekcióban tudjuk tárolni)**. A grafikus alakzatokat ebből a Shape osztályból származtatjuk. Pl. LineShape, RectShape, EditableTextShape.
  - **Probléma**: az EditableTextShape (egy szerkeszthető szövegdoboz teljes logikájának) megvalósítása nagyon-nagyon nehéz, beláthatatlan mennyiségű munka.

Adapter – példa folytatás

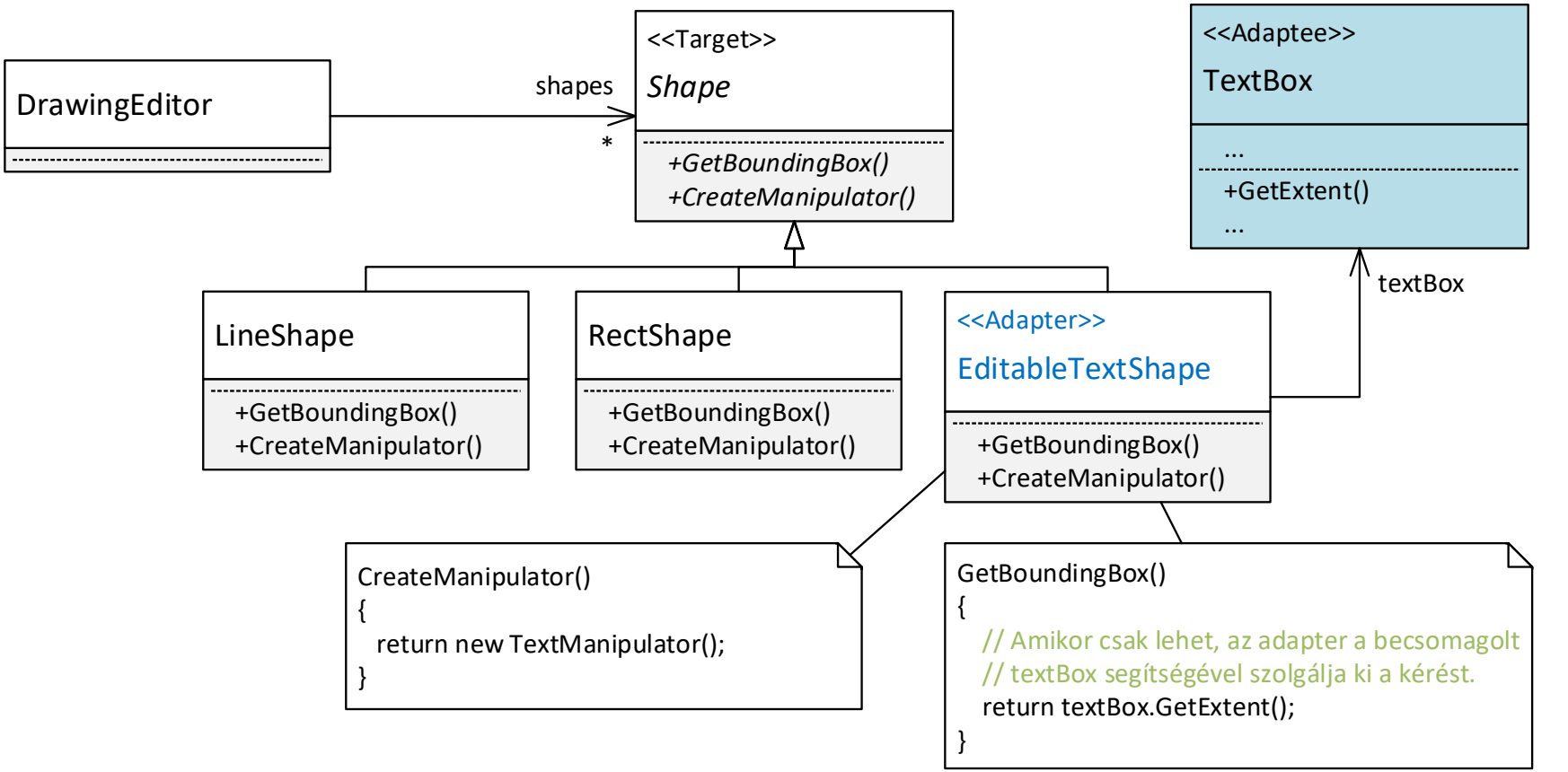
- T.f.h. az alkalmazást egy adott keretrendszerre (pl. UWP) építve írjuk, amiben **már van egy beépített szerkeszthető szövegdoboz** (TextBox osztály), ami tudja mindazt, amit az EditableTextShape-től elvárunk. Jó lenne ezt felhasználni.
- Probléma: a **beépített TextBox osztályt nem tudjuk közvetlenül felhasználni, mert nem megfelelő az interfésze**, ugyanis nem a Shape osztályból származik (emiatt nem tudjuk a többi Shape-el együtt egységesen kezelni). Mivel egy „beépített” osztály, a forráskódját nem is tudjuk módosítani (hogy a Shape osztályból származzon)
- Probléma: nem tudjuk újrafelhasználni a meglévő osztályt!
- Megoldás: Adapter minta használata (Object Adapter vagy Class Adapter)

Kiindulás



- A `DrawingEditor` egy `Shape` listát tárol (heterogén kollekció), a `TextBox` nem tehető bele
- A `GetBoundingBox` művelet egy befoglaló téglalapot ad vissza
- A `CreateManipulator` egy „manipulátor” objektumot, amivel az adott típusú alakzat szerkeszthető (pl. átméretezető, stb., nincs jelentősége a példában)

Példa megoldás Object Adapterrel



Magyarázat

Példa megoldás Object Adapterrel

- Az eredeti tervünknek megfelelően leszármaztatunk az `Shape` osztályból (vagy, ha a `Shape` interfész, implementáljuk azt) → Ez lesz az `EditableTextShape` osztály. Így az EditableTextShape-nek jó lesz az interfésze, a `DrawingEditor` tudja ezt is kezelni.
- Az `EditableTextShape` nem maga valósítja meg a műveletek többségét. Helyette:
  - Példányosít és becsomagol egy `TextBox` osztályt
  - A műveletek többségénél továbbhív a `TextBox` osztályba, delegálja a kéréseket. A példában a `GetBoundingBox` művelet a `TextBox` `GetExtent-et` hívja.
- Így a `EditableTextShape` fel tudja használni a már meglévő komplex logikát (`TextBox` osztály)
- Megjegyzés: a példában a `Shape` egy absztrakt osztály, lehetne interfész is, nincs jelentősége.

Példa megoldás Object Adapterrel

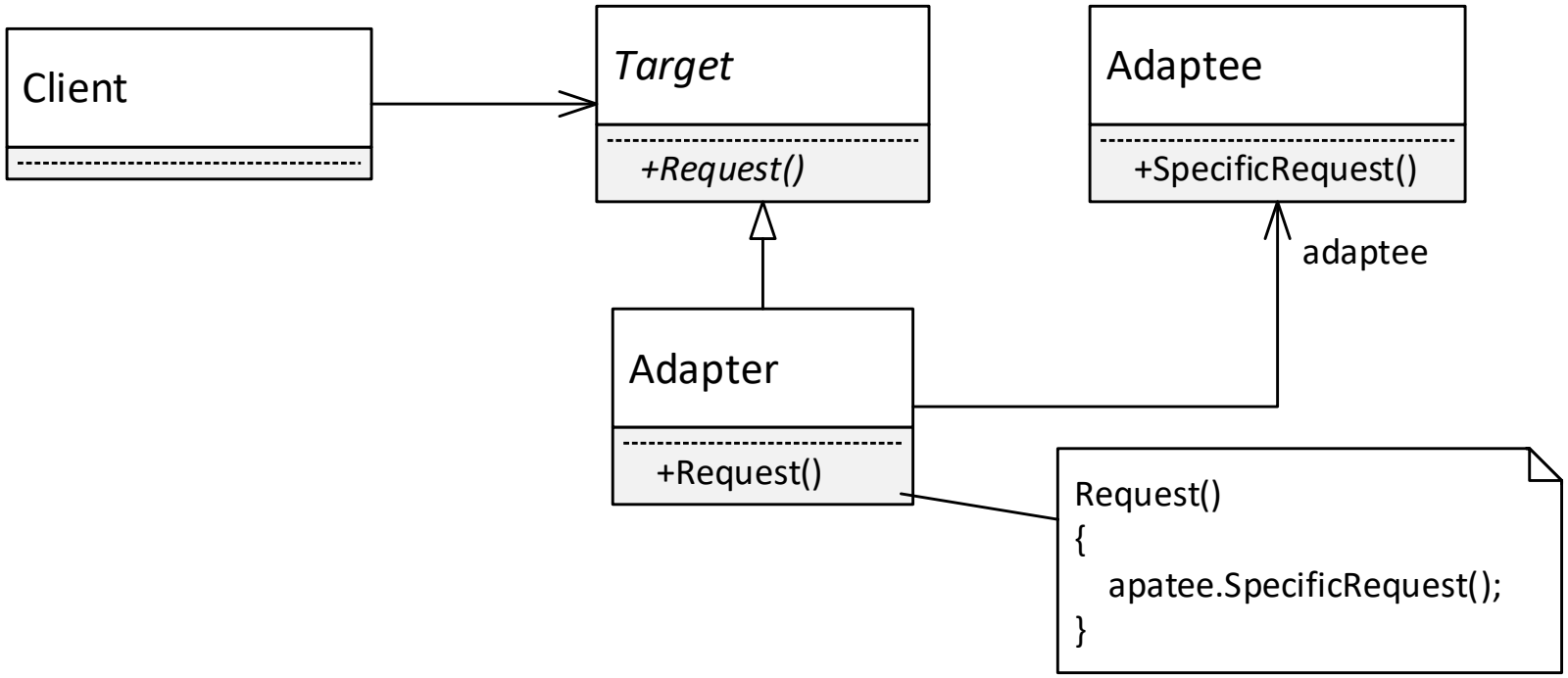
- Kód: lásd DesPattCode/Adapter mappa
- A legfontosabb rész maga az adapter

```
class EditableTextShape : Shape
{
    // A becsomagolt/adaptálandó osztály
    TextBox textBox;

    // ...

    protected override Rect GetBoundingBox()
    {
        // Ez a lelke, ahol csak lehet, továbbítjuk a kérést
        // a becsomagolt/adaptálandó osztálynak, felhasználjuk
        // a kódját
        return textBox.GetExtent();
    }
}
```

Adapter

- Adapter minta általánosságában
- Két változat
  - Object Adapter
  - Class Adapter

Object Adapter struktúra

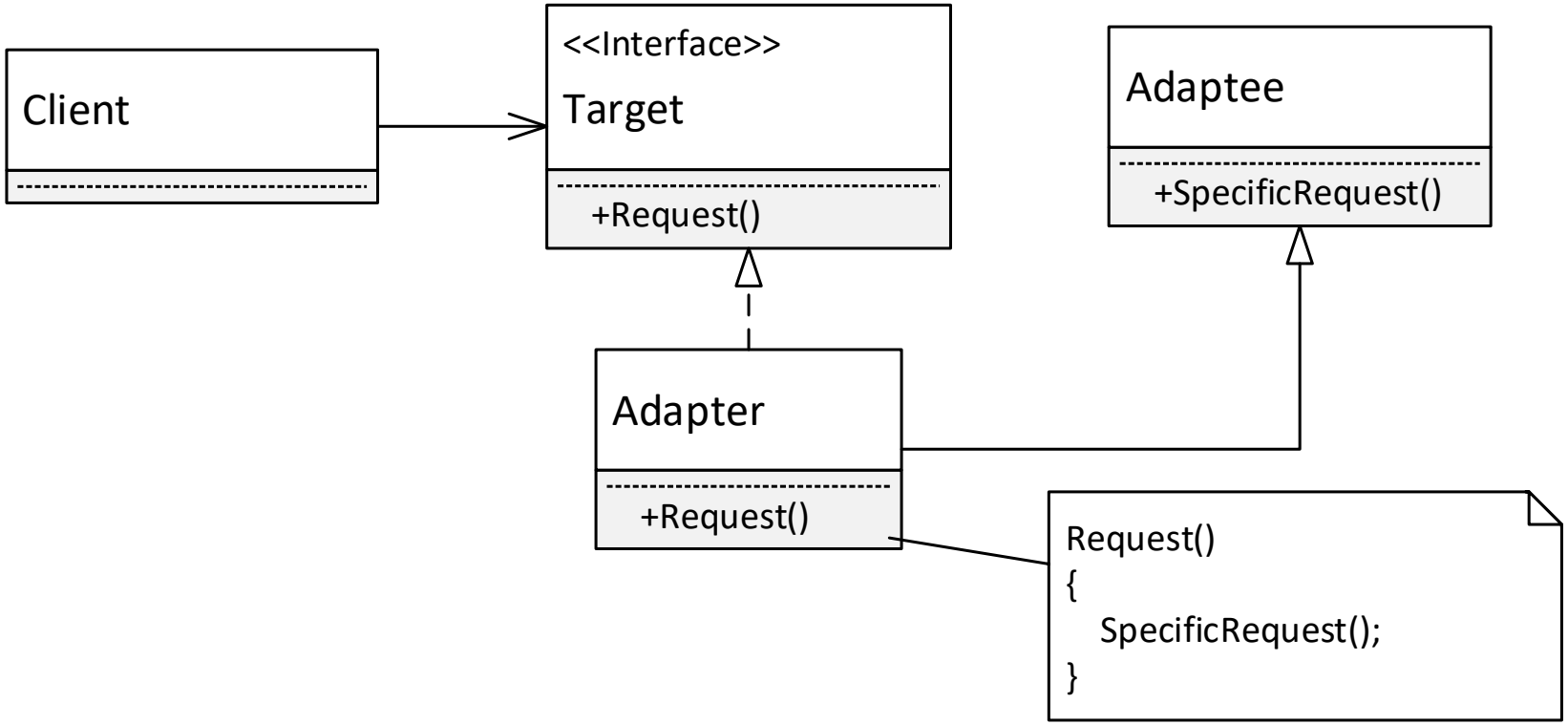


Magyarázat

Object Adapter magyarázat

- **Adaptee**: Az adaptálandó osztály, melynek nem megfelelő az interfésze (emiatt nem tudjuk az adott környezetben triviális egyszerűséggel felhasználni).
- **Adapter** (*EditableTextShape*) : Illesztő, az Adaptee interfészt a Target interfésszé „konvertálja”. Tartalmaz egy hivatkozást egy adaptee objektumra (becsomagol egyet). A műveletei, ahol csak tehetik, a becsomagolt adaptee műveleteit hívják.
- **Target** (*Shape*): Interfész, amit a kliens használ. A gyakorlatban lehet interfész vagy absztrakt osztály is.

Class adapter struktúra



Alapja: tartalmazás (becsomagolás) helyett leszármaztatás.

Magyarázat

Class Adapter magyarázat

- A szerepek alapvetően megegyeznek az Object Adapterével, ami különbség:
  - **Az Adapter nem becsomagolással használja fel az Adaptee kódját, hanem leszármaztatással.** Az Adapter műveletei, ahol csak tehetik, **az ős Adaptee műveleteit hívják**.
  - Mivel a legtöbb nyelv (pl. Java, C#) nem támogatja a többszörös öröklést, a Target itt csak interfész lehet, ősosztály nem!
  - Kevésbé rugalmas, mint az Object Adapter, általában az Object Adapter változatot preferáljuk!

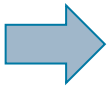
Adapter összefoglaló

- Használjuk: **ha nem tudunk valamilyen osztályt újrafelhasználni egy környezetben, mert nem jó az interfésze**
- Megjegyzések
  - (Az Object Adapter esetében nem minden esetben szükséges tagváltozóként tárolni az Adaptee-t, ritkább esetben az Adapter művelete maga példányosítja majd el is dobja az Adaptee objektumot. Ekkor nem asszociáció, hanem csak függőség van az Adapter és az Adaptee között.)

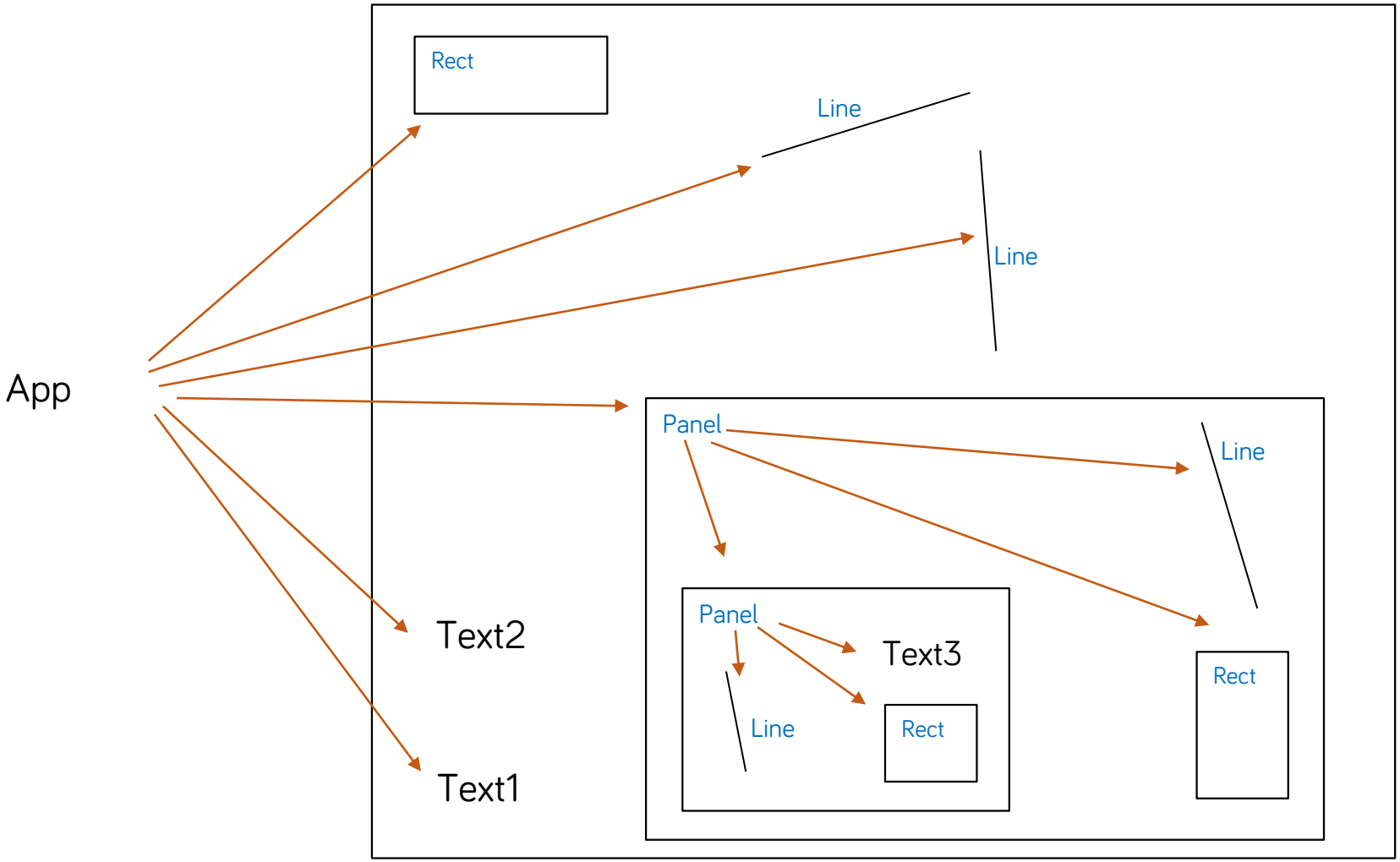
Composite (Összetett)

Composite

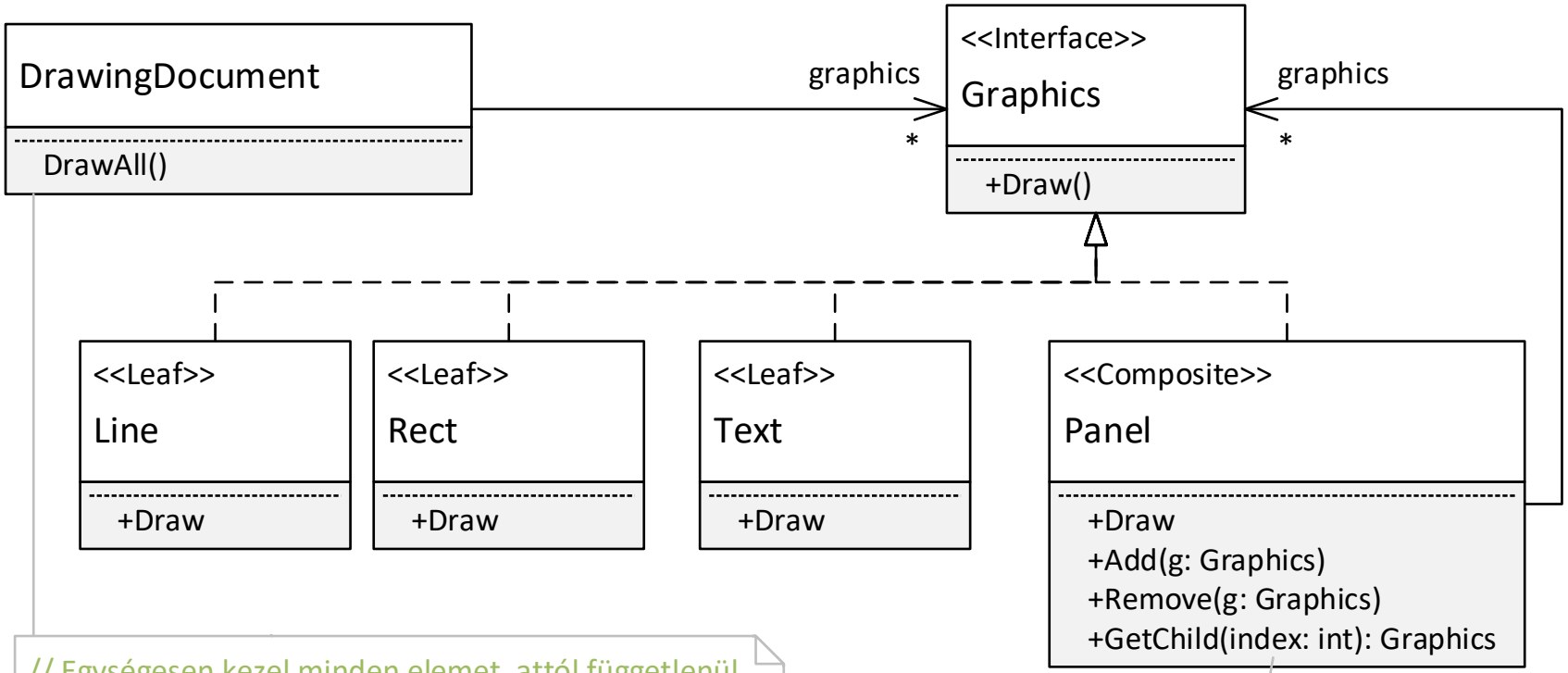
- Célja
  - Rész-egész viszonyban álló objektumokat fastruktúrába rendezi
  - A kliensek számára lehetővé teszi, hogy az egyszerű és összetett (kompozit) objektumokat egységesen kezelje
- Példa
  - Olyan grafikus alkalmazás, amely lehetővé teszi elemi és összetett grafikus objektumokat tartalmazó rajzok létrehozását
  - A Panel egy olyan összetett grafikus elem, mely grafikus alakzatokat tartalmaz (tetszőlegeseket, akár más Panel objektumok is lehetnek rajta!)



Composite példa



Composite példa




```
// Egységesen kezel minden elemet, attól függetlenül,
// hogy elemi (levél) vagy összetett: Draw()-t hív rajta
DrawAll()
{
   foreach (g in graphics) g.Draw()
}
```

```
// Minden tartalmazott gyerekre Draw()-t hív
Draw()
{
   foreach (g in graphics) g.Draw()
}
```

Magyarázat

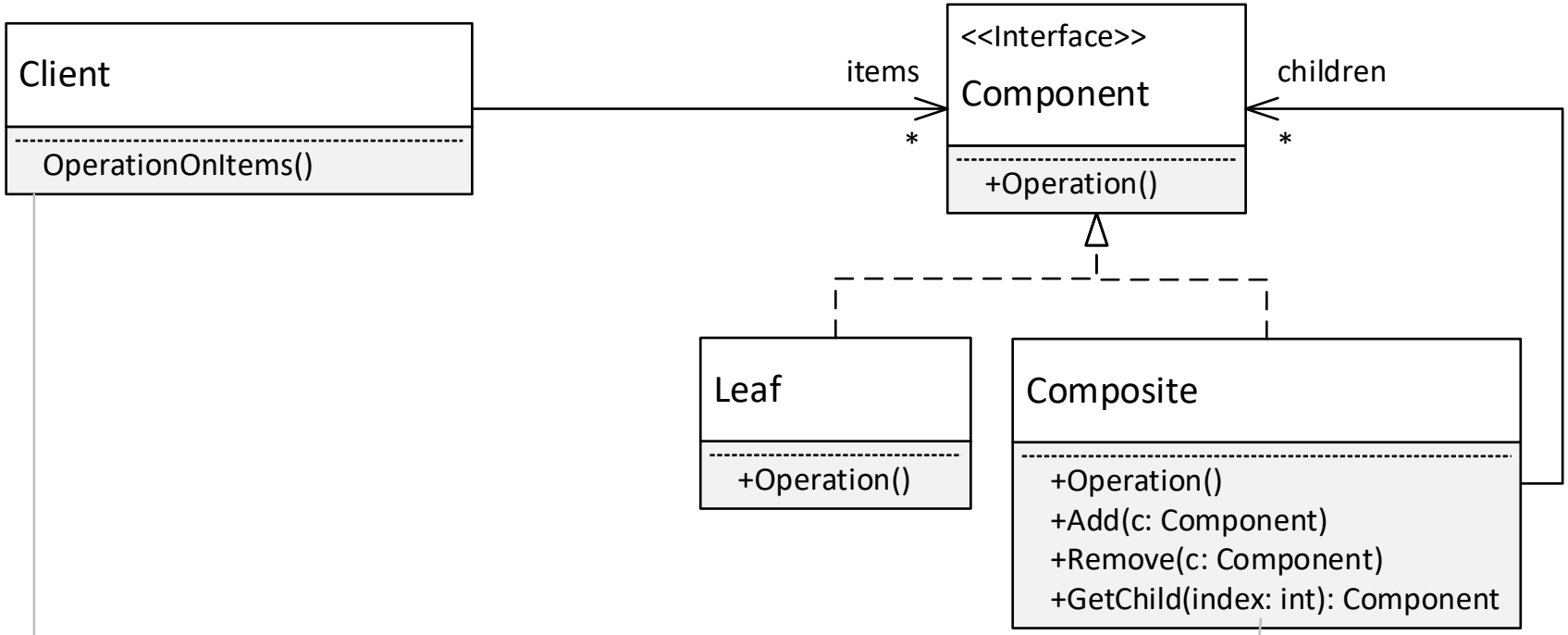
Composite példa magyarázat

- **DrawingDocument**: egy olyan dokumentumot reprezentál, mely különböző grafikus alakzatokat tárol és kezel
- Ezen grafikus alakzatok lehetnek
  - Elemiek (levél, leaf), mint pl. Line, Rect, Text.
  - Összetettek, mint pl. a Panel (mely maga is tartalmazhat elemi és összetett alakzatokat).
- **Graphics**: a grafikus alakzatok közös interfésze/őse, attól függetlenül, hogy elemi vagy összetett az alakzat

Composite példa magyarázat

- A Composite minta egyik alapelve, hogy **egységesen kezeli az elemi (`Line`, `Rect`, `Text`) és az összetett (`Panel`) objektumokat**. Ez a példában két pontban is megjelenik:
  - A `DrawingDocument` **közös gyűjteményben tárolja az elemi és összetett objektumokat** (ez a graphics tag). Megteheti mert az összetett `Panel` is implementálja a `Graphic` interfészt.
  - A `DrawAll` műveletben a kirajzoló kód **nem különböztetni meg – vagyis egységesen kezeli - az elemi és összetett objektumokat** (nincsenek típus szerint leválogatva): mindre egységesen a `Draw()` műveletet hívja.
- A összetett `Panel Draw` művelete az általa tartalmazott alakzatokat rajzolja ki, pont ez volt a célunk!
- Kód: lásd DesPattCode/Composite mappa

Composite általánosságában




```
// Egységesen kezel minden elemet, attól függetlenül,
// hogy elemi (levél) vagy összetett: Operation()-t hív rajta
OperationOnItems()
{
   foreach (i in items) i.Operation()
}
```

```
// Minden tartalmazott gyerekre Operations()-t hív
Operation()
{
   foreach (c in children) c.Operation()
}
```

Composite

- Használjuk, ha
  - Objektumok rész-egész viszonyát szeretnénk kezelni (jellemzően fastruktúrában)
  - A kliensek számára el akarjuk rejteni, hogy egy objektum elemi objektum vagy kompozit objektum: bizonyos műveletek szempontjából egységesen szeretnénk kezelni őket

Composite megjegyzések

- A gyerekek kezelése nem tud egységes lenni!
  - Hiszen az csak az összetett (példánkban a Panel) osztály objektumaira értelmezett (Add, Remove, GetChild)
  - A kliens pl. típusellenőrzéssel tudja megnézni, hogy az adott elem összetett-e (pl. C#-ban az „is”, Java-ban az „instanceof” operátorral), pl.:
    - if (item is Composite) …
  - Az Add, Remove, GetChild műveletek hívásához a kliensnek a kompozit osztályra/interfészre kell castolnia a hivatkozását

Tervezési minták összefoglaló

További tervezési minták

- A klasszikus „GoF” minták közül is kimaradt pár:
  - Prototype, Builder, Bridge, Mediator, Chain of Responsibility, Visitor, Decorator, Iterator, State, stb.
- Számos nem általános tervezési minta létezik, pl.
  - Elosztott, konkurens rendszerekre jellemző minták
    - Szolgáltatás hozzáférés
    - Konfiguráció
    - Esemény kezelés
    - Szinkronizáció
    - Konkurencia
  - Valós idejű rendszerekre jellemző minták
  - Vállalati 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 mintát
  - Hosszútávon legalább arra emlékezzünk, hogy egy adott problémakörben mely mintákat lehet jól használni