Tervezési Minták:

Létrehozási minták:

- Objektumok létrehozásával kapcsolatos
- A sima new-zás néha nem elég jó, rugalmatlan

Singleton:

Cél:

- Biztosítja, hogy egy osztályból 1 példány legyen, és azt globálisan el lehessen érni
  - private, static változóban tárolja az instanceot
  - statikusan el lehet azt érni getterrel vagy propertyvel
  - privát vagy protected ctor, nem lehet new-zni

Példa:

```
class WindowManager
{
    // Tárolja az egyetlen példányt, kezdetben null.
    private static WindowManager instance;

    // Glogális hozzáférést biztosító statikus művelet
    public static WindowManager GetInstance()
    {
        // Az egy példányt az első hozzáférés alkalmával hozzuk létre
        if (instance == null)
            instance = new WindowManager();
        return instance;
    }
    // Védett konstruktor
    protected WindowManager() { }

    // Az osztály egyik művelete
    public void DoSomething() { /*...*/}
}

…
// Valahol a kódban hozzáférünk hozzá a WindowManager singleton példányhoz
WindowManager.GetInstance().DoSomething();
```

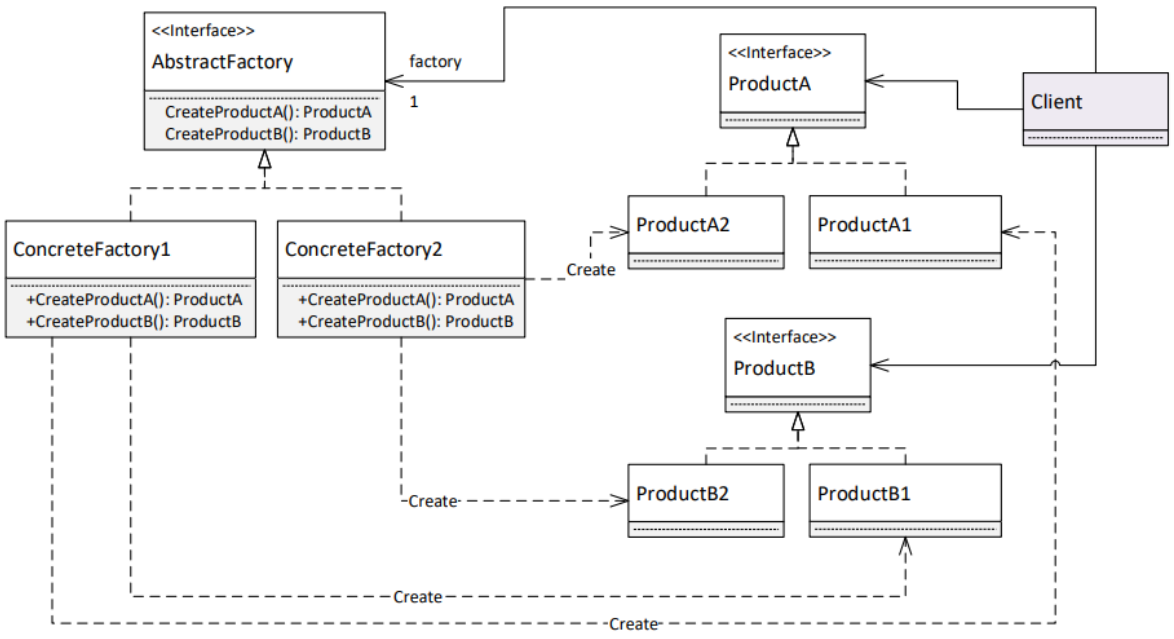
Sidenote: Azért ne vigyük túlzásba, plusz néha lehet antipattern, mert a GetInstance mindig az adott típussal tér vissza, nem lehet dummyt csinálni belőle pl unit tesztekhez.

Abstract Factory:

Cél:

- Interfészt biztosít ahhoz, hogy egymással összefüggő objektumok családjait hozzuk létre konkrét osztály specifikálása nélkül.
- Létrehozás egy interfészen keresztül történik, nem függ a létrehozott objektumok konkrét típusától.
    - Nem drótozzuk be a létrehozásokat, hanem az abstract factorynak adjuk ennek felelősségét. Így bármikor lecserélhető lesz.

Általánosan:



Mikor használjuk:

- Amikor a rendszernek függetlennek kell lennie az általa létrehozott dolgoktól
- Több termékcsaláddal kell együttműködnie
- Szorosan összetartozó termék objektumok családjával kell dolgoznia, és ezt akarjuk kikényszeríteni
    - Példa:
        - Windowsos és MacOS-es GUI elemek létrehozása.
        - Windowsos GUI-hoz ne tartozhasson MacOS-es srcollbar pl.

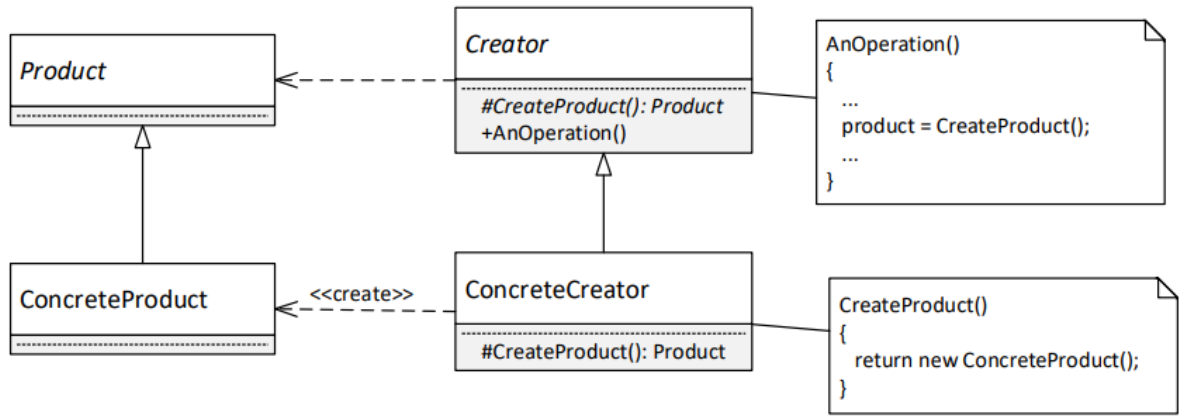
Pros	Cons
Elszigeteli a konkrét osztályokat	Új termék hozzáadása nehéz, ekkor az egész hierarchiát módosítani kell
Termékcsaládok könnyen cserélhetők	Közös őssel néha elkerülhető
Elősegíti a termékek közti konzisztenciát	

Factory method:

Célja:

- Interfészt definiál objektum létrehozására, de a leszármazott osztályra hagyja a konkrét osztály eldöntését.
- Lehetővé teszi, hogy a leszármazottra bízzuk a konkrétumokat
- ~Virtuális konstruktornak is szokás nevezni(Ezt egy prog2 hallgató meg ne tudja)
- Igazából egy Template Method létrehozási mintaként.

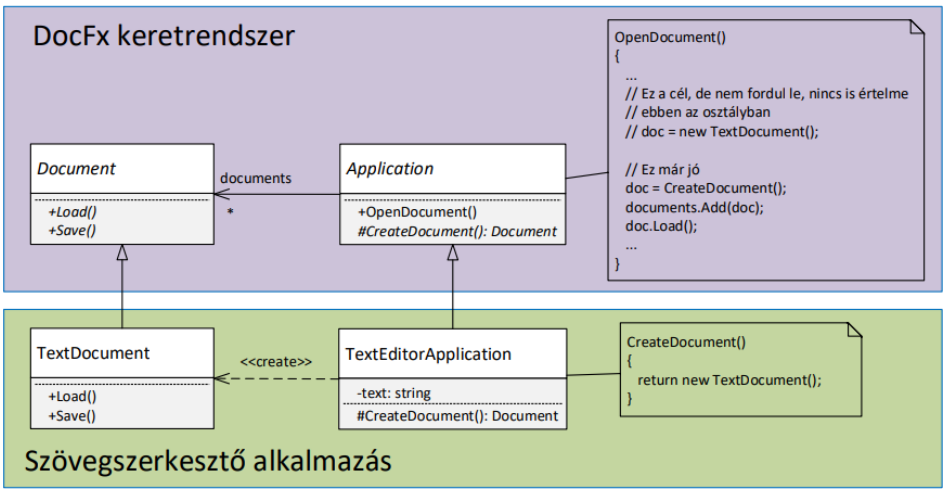
Általáosan:



Magyarázat:

- Product: Absztrakt termék osztály
- ConcreteProduct: Konkrét termék, amit a Creatorban létre akarunk hozni.
- Creator: Absztrakt osztály, ami egy általa nem ismert product leszármazottat akar létrehozni.
- ConcreteCreator: Konkrét létrehozó osztály, Creator CreateProduct metódusát felüldefiniálja, visszatér egy abban létrehozott ConcreteProducttal.

Példa:



Használjuk, ha:

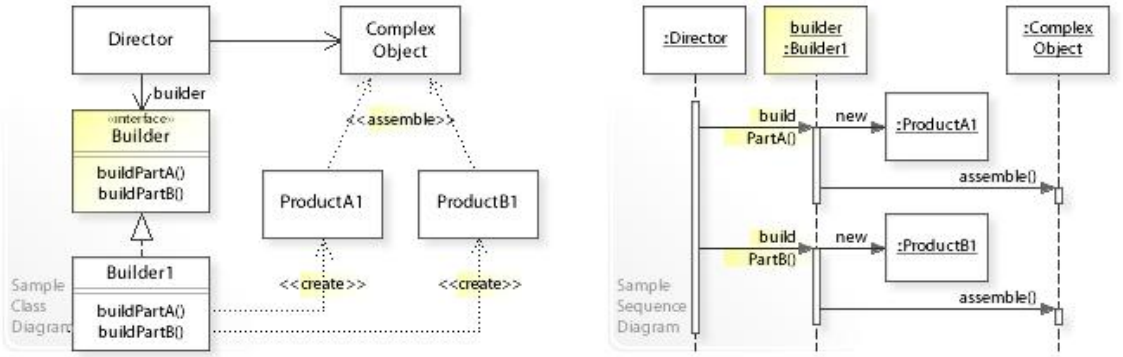
- Egy osztály nem látja előre annak az objektumnak az osztályát, amit létre akar majd hozni.
- Ha az osztály azt szeretné, hogy a leszármazottai határozzák meg a létrehozandó osztályt.

Builder:

Célja:

- Az Építő tervezési minta célja az összetett objektumok kialakításának és reprezentációjának szétválasztása. Ezáltal ugyanaz az építési folyamat különböző megvalósításokat eredményezhet.

Általánosan:



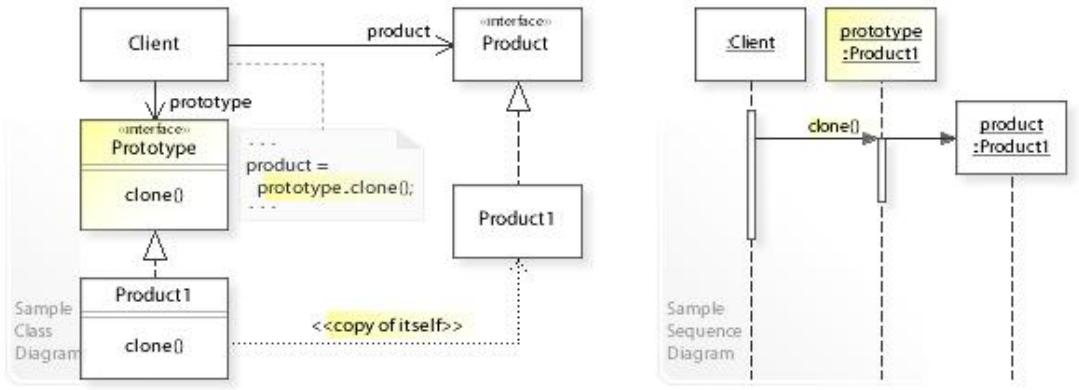
Magyarázat:

- Director: Nem ő hozza létre az objektumot, csak hívja a Buildert, mert a Builder felelőssége ez
- Builder: Absztrakt gyártó interfész.
- Builder1: Ez a konkrét gyártó osztály. Polimorfizmust kihasználva más-más objektum létrehozására szolgál.
- Product: A komplex objektum egy része, amit külön létrehozunk

Pros	Cons
Könnyen megváltoztatható a termék belső reprezentációja	Minden termékhez(product) kell egy külön Builder
Egységbezárja a létrehozásés a reprezentálás kódját.	A builder osztályoknak mutábilisnak kell lenni
Irányítást biztosít a létrehozás folyamata fölött	Nem támogatja a Dependency injectiont

Prototype:

- Factory methodhoz nagyon hasonló tervezési minta, de új objektum létrehozása helyett klónozza magát.
- https://stackoverflow.com/questions/20043279/factory-vs-prototype-what-to-use-when



Object Pool:

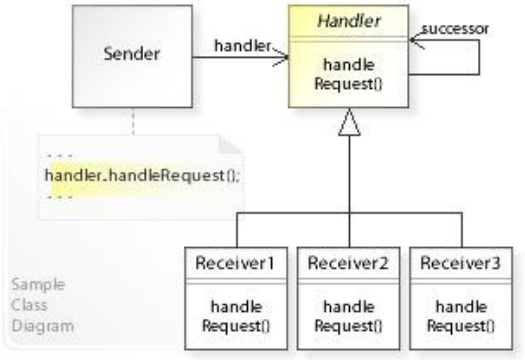
- Egy olyan létrehozási minta, ami nem hoz létre új objektumokat, hanem a kliens előre létrehozott objektumokból választ ki egyet, használja, majd elpusztítása helyett visszateszi a helyére azt.
- Általában teljesítményi okokból használják.
- Ha sok olyan objektummal kell dolgozni, amiket létrehozni, elpusztítani drága, viszont általában rövid időre kellenek, akkor érdemes használni.
- C#-ban a ThreadPool például így működik (Szoftvertechnikák)

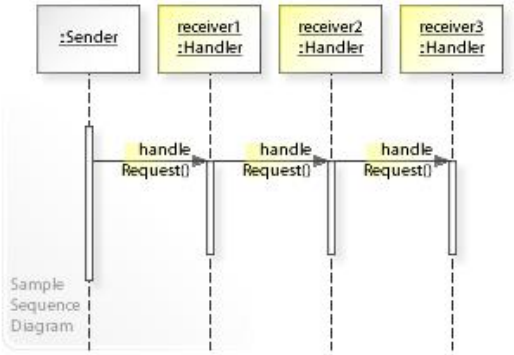
Viselkedési minták:

Chain of responsibility:

- Ez a minta arra szolgál, hogy egy kérés feldolgozásánál lévő hosszú if-else ágat vált fel objektumorientált, polimorfizmust kihasználó osztályokra.
- Akár több fogadó is tudja a kérést teljesíteni.
- Ezzel a patternnel a kérés küldője és fogadója között csökken a csatolás.

Általánosan:





Magyarázat:

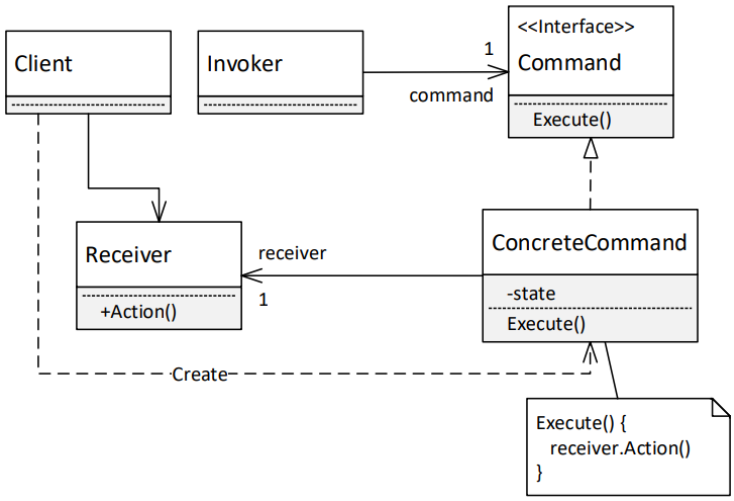
- Sender lát egy Handler interfészt. Elküldi neki a kérést.
- A Handler interfészt megvalósítja több Receiver osztály, akik vagy teljesítik a kérést, vagy továbbküldik a következő Receivernek.
- Ebbe a láncba bármikor felvehető új Receiver.

Command:

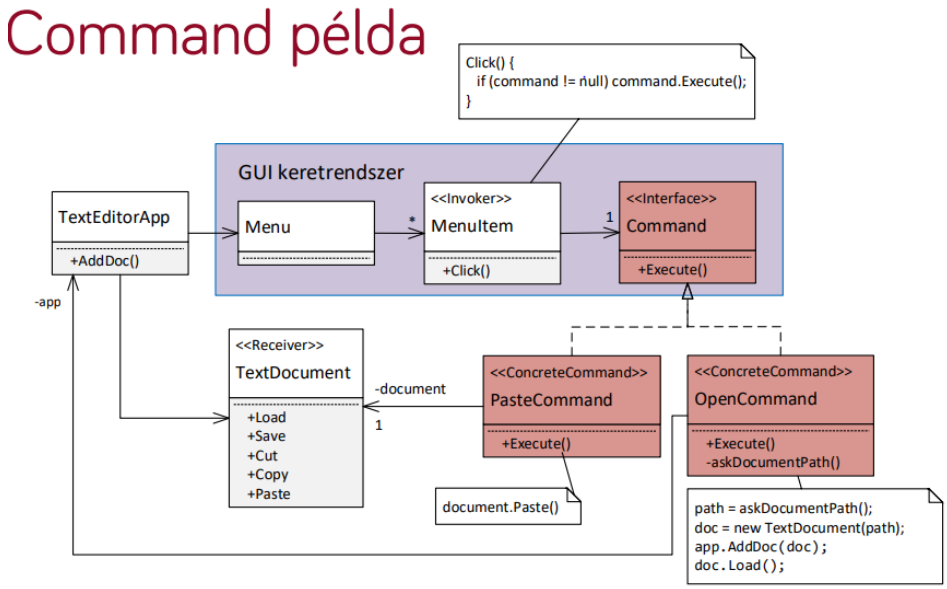
Cél:

- Egy kérés objektumként való egységbezárása
- Kérés függvényhívás helyett objektum
    - Ez lehetővé teszi
        - kliens különböző kérésekkel való felparaméterezését
        - kérések sorba állítását
        - naplózását
        - visszavonását (undo)

Általánosan:



Példa:



Magyarázat:

- Vannak a menüelemek, amik egy adott commandot akarnak végrehajtani
- Az adott menüelemet adott commandra irányítunk rá, hogy azt futtassa kattintás esetén
- Így
    - Eltérő menüelemek is irányíthatók ugyanarra a parancsra
    - Különböző menüelemek irányíthatók különböző parancsokra
- Elválasztja a parancskiadó objektumot a parancsvégrehajtótól

Megjegyzés:

- A Command.Executéba mennyi logikát teszünk?
    - Akár benne lehet minden logika
    - De akár delegálhatja a parancsvégrehajtást egy másik classnak

Mikor használjuk:

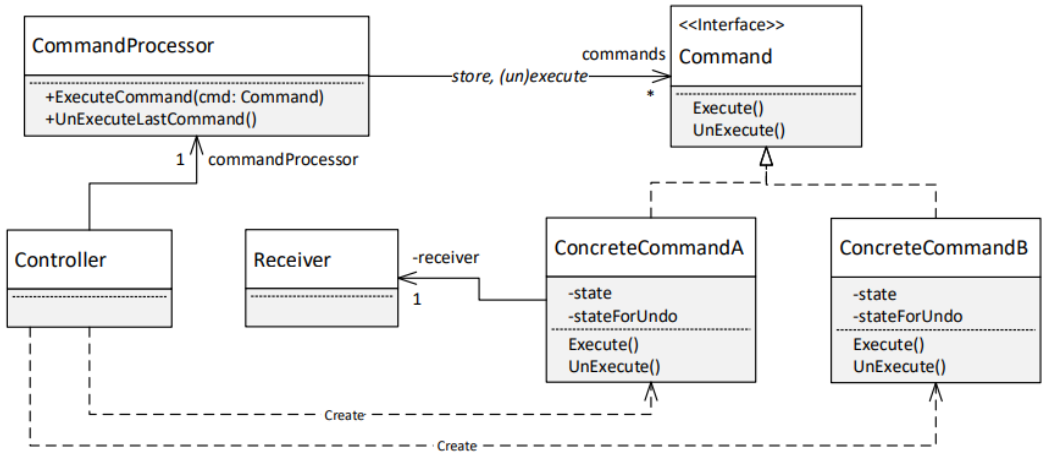
- Callback függvények helyett OO programokban
- Objektumspecifikus kódot tudunk így futtatni
- Visszavonás támogatása
    - Eltároljuk az előző állapotot a Commandban, ehhez majd a Command Processor segít
- Kérések különböző időben való kiszolgálása

Command Processor:

Cél:

- Támogatja a parancsok visszavonását(Undo vagy UnExecute vagy Revert)
- Command Processor osztály bevezetése
    - Tárolja a futtatott command objektumokat, hogy visszavonás esetén rendelkezésre álljon az
    - Rajta keresztül futtatjuk és vonjuk vissza a parancsokat

Általánosan:



Undo lépései:

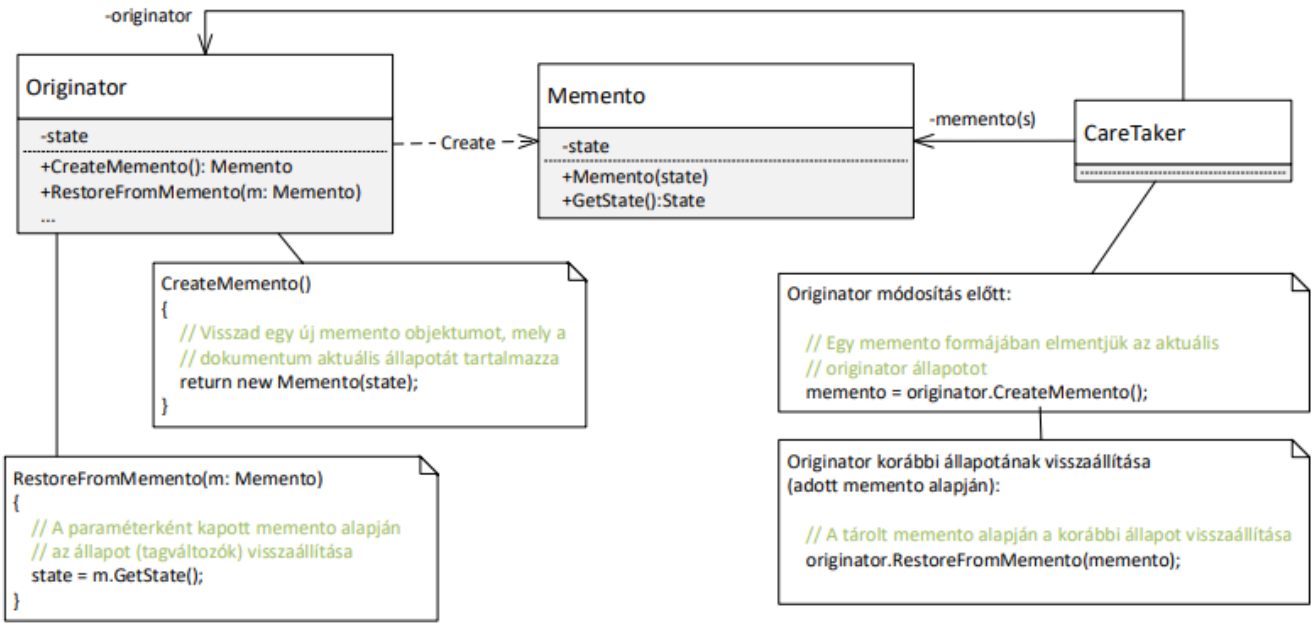
- Felhasználó kéri az UnExecuteLastCommandot
- Az app controllere fogadja a kérést, továbbadja a CommandProcessornak
- Ez kiveszi a legutoljára eltárolt Commandot a Command gyűjteményből
- Erre a Commandra meghívja az UnExectue metódusát
- Így a Commandnak lefut a visszacsináló kódja

Memento:

Cél:

- Egységbezárás megsértése nélkül a külvilág számára elérhetővé tenni az objektum belső állapotát
    - Nem egyenlő azzal, hogy a tagváltozókat publiccá tesszük
    - Így az objektum állapota később visszaállítható
- Elmenteni az objektum állapotát Undo parancsnhoz
    - Enélkül is használható, de célszerű Command/Command Processorral együtt Mementot is használni
- Objektum adott állapotát egy Memento objektumba csomagoljuk, és azt tesszük elérhetővé publikusan

Általánosan:

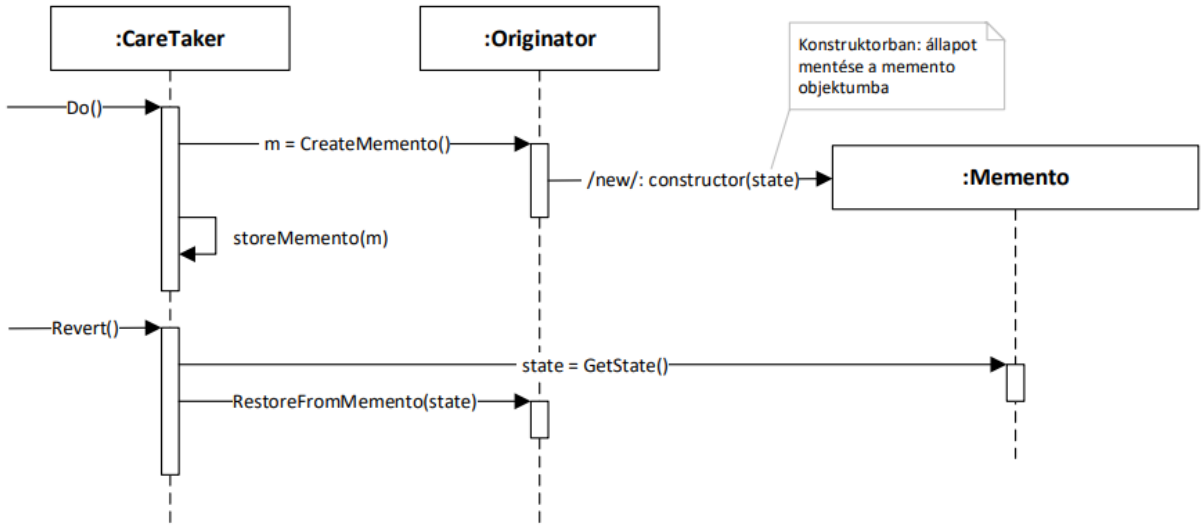


Magyarázat:

- Originator: Az ő állapotát kell tudni visszaállítani
    - CreateMemento visszaadja a state állapotot egy Memento objektum formájában
    - RestoreFromMemento visszaállítja az állapotot a Memento alapján

- Memento: Az Originator állapotát tárolja, és elméletileg csak az Originator férhet hozzá
- CareTaker: Nyilvántartja a Mementokat

Szekvenciadiagram:



Használjuk, ha egy objektum állapotát később vissza kell állítani, és ennek támogatásához meg kéne sérteni az egységbezárás elvét.

Előny: Megőrzi az egységbezásrást

Hátrány: Sokszor erőforrásigényes, pl teljes dokumentumok többszöri lementése

Observer:

Cél:

- Lehetővé teszi, hogy egy objektum értesítést küldjön más objektumoknak az állapotának változásáról.
- Lehetővé teszi, hogy objektumok értesíteni tudják egymást állapotuk megváltozásáról anélkül, hogy függőség lenne köztük konkrét osztályaiktól.

Magyarázat:

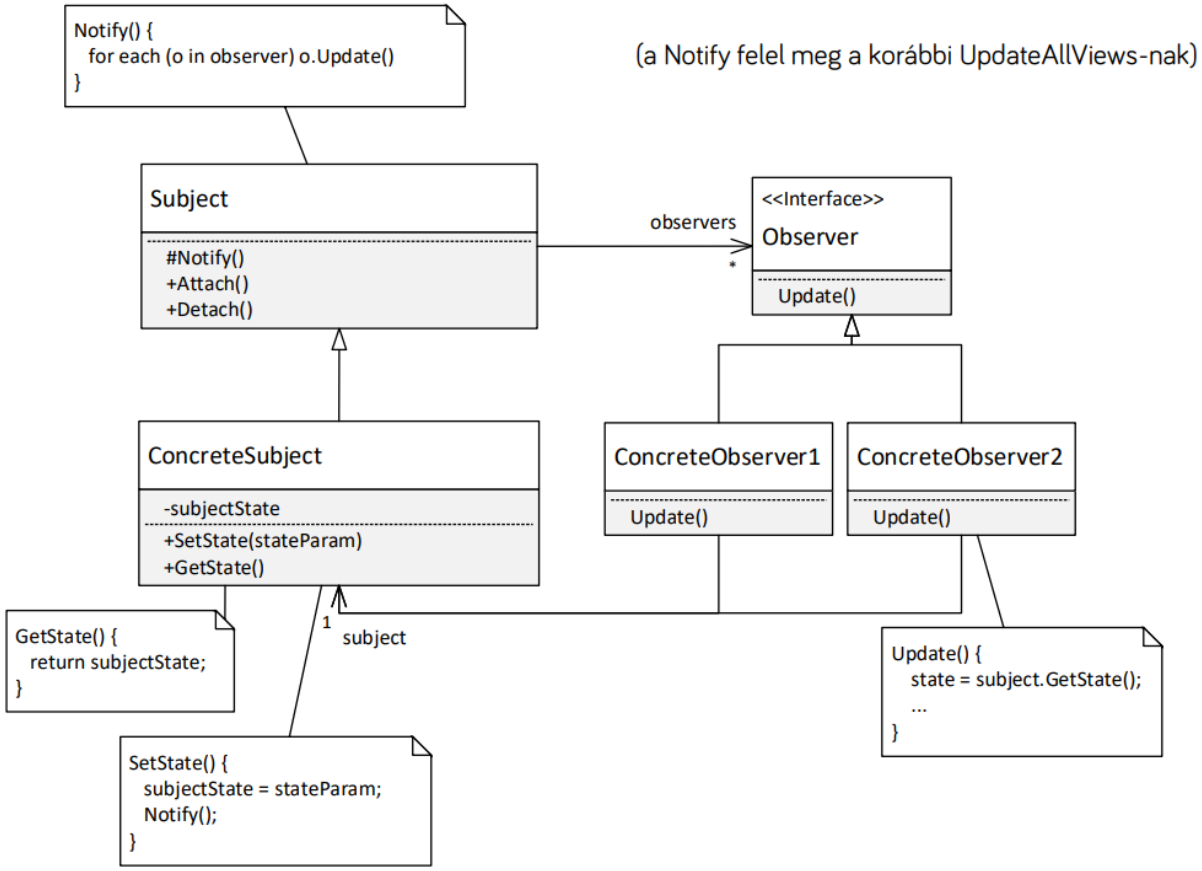
- Az adatoknak egyetlen közös, hiteles forrása legyen, adatokat és azon értelmezett műveleteket tároljon.
- Lehessen hozzá view-eket beregisztrálni(ezek az observerek)
- Ha valamelyik view változik, a dokumentum adatait megváltoztatja, majd a dokumentum szól a változásról a többi viewnak.
- A view ennek hatására lekérdezi a dokumentumot és frissíti magát.
- A dokumentum a vieweket csak egy közös ősön keresztül látja.

- Ez maga a document-view architektúra.

Előnyök:

- A rendszer könnyen kiterjesztehető új view osztályokkal.
  - Semmit nem kell ehhez módosítani.
- Egyszerű mechanizmus arra, hogy az összes view konzisztens nézetét mutassa az adatoknak.
- A document csak egy view listát tárol, így a modell független a viewtől.
- A document újrafelhasználható.

Általánosan:

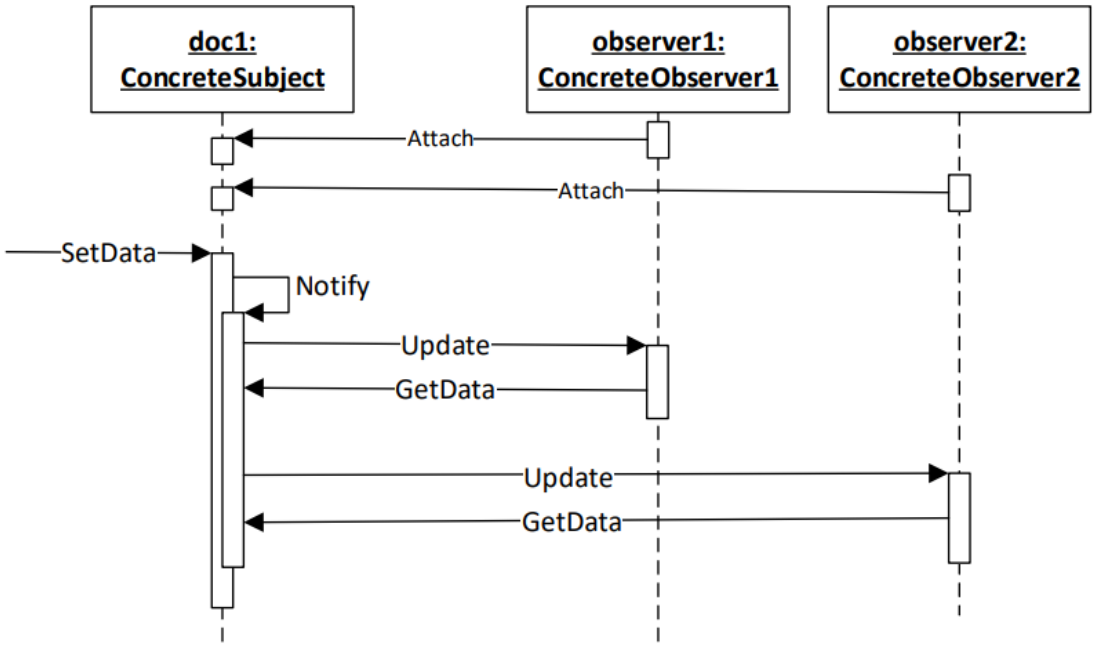


Magyarázat:

- Subject:
  - Tárolja a beregisztrált observereket.
  - Lehetőséget ad azok ki és beregisztrálására.
  - Tartalmazza az observerek közös ősét.
  - Gyakorlatban nem mindig jelenik meg
- Observer:
  - A subjectben bekövetkező változások iránt érdeklődő osztályok őse.

    - Interfészt definiál számukra.
- ConcreteSubject:
    - Az observerek számára érdekes állapotot tárol.
    - Értesíti az observereket állapota változásakor.
- ConcreteObserver:
    - Pointere(Referencia C#-ban) van a megfigyelt objektumra.
    - Olyan állapotot tárol, amit a ConcreteSubjecttel konzisztensen kell tárolni.
    - Implementálja az Observer interfészt.
    - Update esetén frissíti a saját állapotát.

Szekvenciadiagram a működésről:



Használjuk, ha:

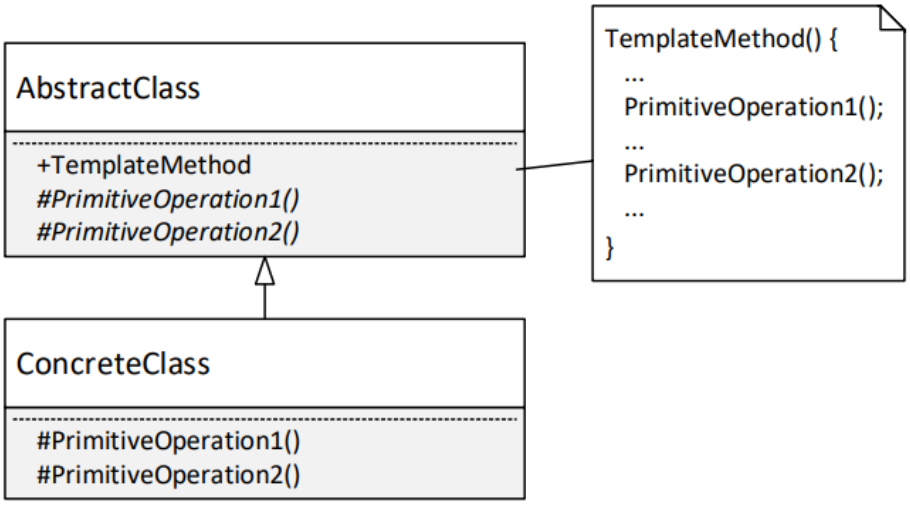
- Egy objektum változása más objektumokat is megváltoztat, és nem tudjuk hányat, esetleg ez a szám változhat is.
- Egy objektumnak értesítenie kell más objektumokat az értesítendő objektum szerkezetére vonatkozó feltételezése nélkül.

Template Method:

Cél:

- Egy műveleten belül algoritmus vázat definiál és az algoritmus bizonyos lépéseinek implementálását a leszármazott osztályra bízza.
- Egy osztály bővítése absztrakt/virtuális függvények segítségével
  - Változatlan részek ősosztályba
  - Változó részek leszármazottba, ősben virtuális/absztrakt metódus

Általánosan:



Példa: (itt a Run a TemplateMethod, compressionok az Operationok)

```
…
public void Run()
{
    byte[] inputData;
    InitCompression(); // Tömörítés inicializás, impl. függő
    try
    {
        while ((inputData = readData()) != null)
        {
            // Tömörítés, impl. függő
            byte[] compressedData = CompressData(inputData);
            processCompressedData(compressedData);
            if (IsCancelled()) // Cancel vizsgálat, impl. függő
                return;
        }
    }
    finally
    {
        CloseCompression(); // Tömörítés lezárás, impl. függő
    }
}
…
```

Következmények:

- Lehetővé teszi, hogy az invariáns részek egy helyen legyenek, a változó részek külön-külön.
- Kódduplikálás elkerülése megoldható(DRY elv).
- Lehetővé teszi kiterjesztési pontok létrehozását a kódban.

Pros	Cons
Kódduplikáció elkerülése	Futás közben nem cserélhető, rugalmatlan
Új viselkedésre bővíteni könnyű	Minden kombinációhoz külön osztály kell, ezért sok lehet egy idő után

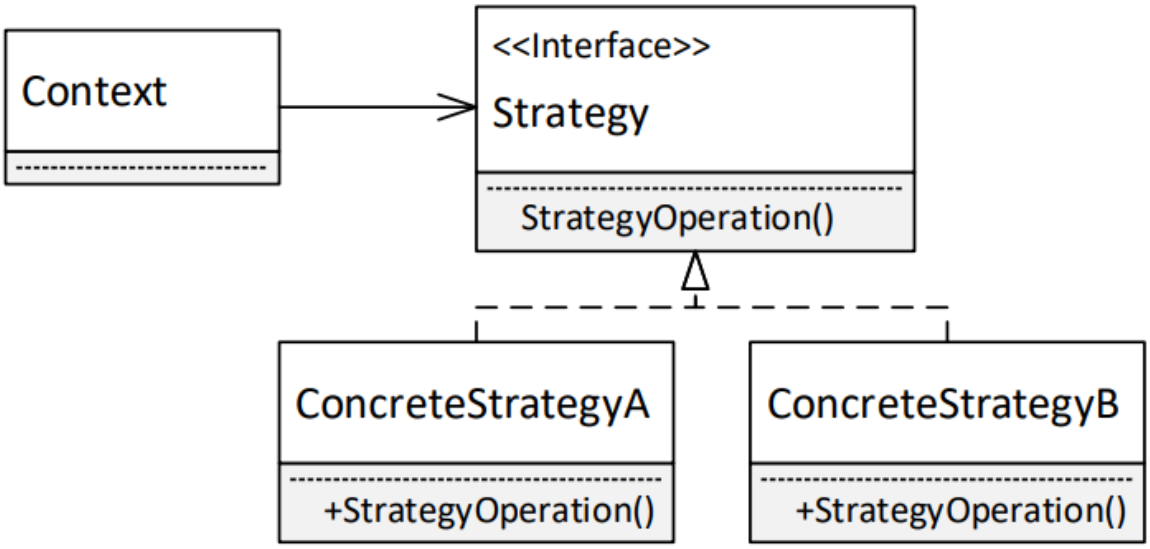
- Egyszerűbb esetekben használjuk általában, vagy keretrendszereknél

Strategy:

Cél:

- Algoritmusok egységbe zárása és egymással kicserélhetővé tétele
- Mindenhol, ahol lecserélhetővé, bővíthetővé akarunk valamit tenni, külön strategy hierarchiát csinálunk.
- Delegál, nem leszármaztat (vagyis a delegált osztály származik majd le)

Általánosan:



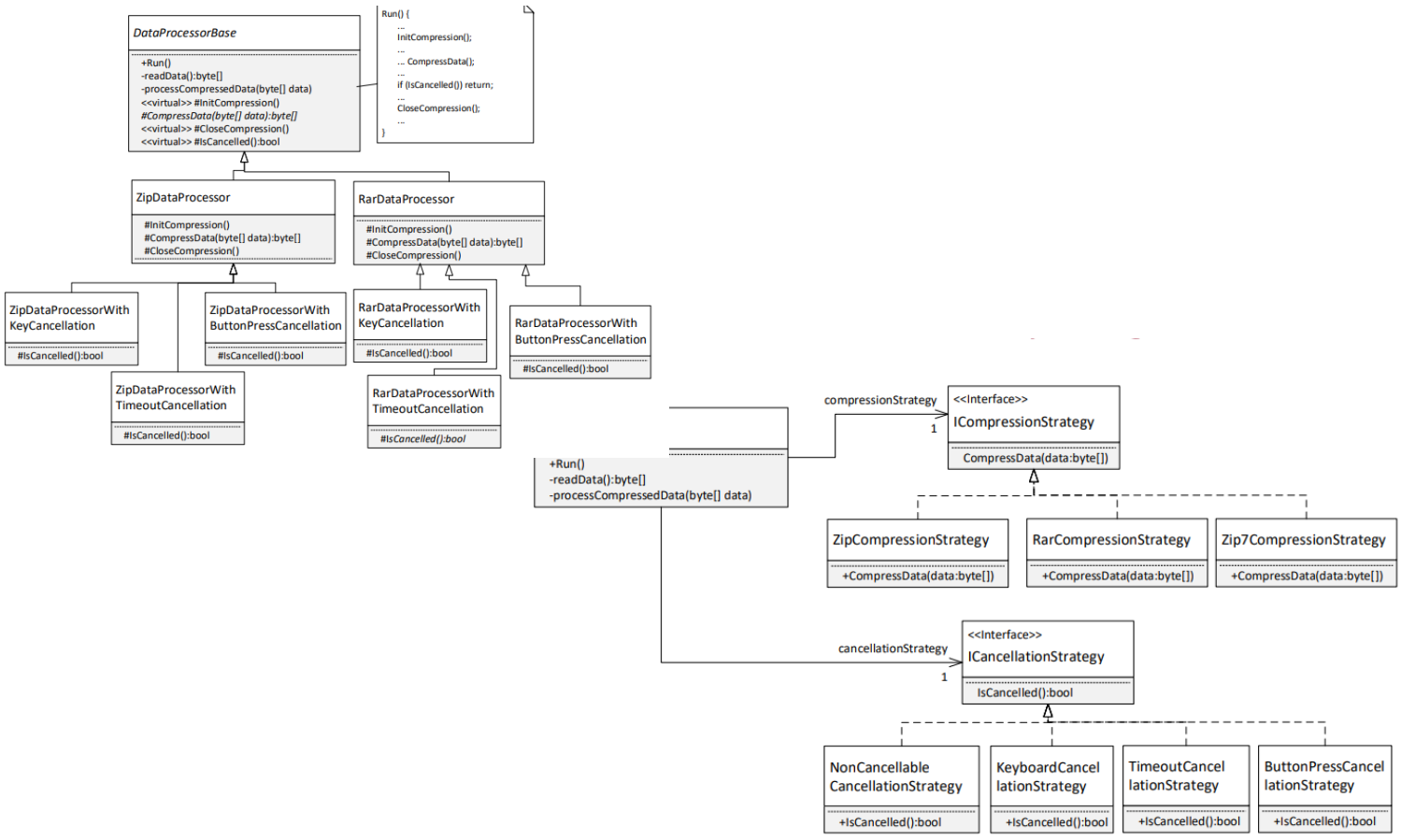
Példa: (vagy igazából hozhatnám a projlabos Jégmező játékot is)

```
class DataProcessor
{
    ICompressionStrategy compressionStrategy; // Aktuális cancel stratégia/viselkedés
    ICancellationStrategy cancellationStrategy; // Aktuális tömörítési stratégia/viselkedés

    public DataProcessor(ICompressionStrategy compressionStrategy,
            ICancellationStrategy cancellationStrategy)
    {
        …
        // Eltároljuk a paraméterként kapott stratégiákat
        this.compressionStrategy = compressionStrategy;
        this.cancellationStrategy = cancellationStrategy;
    }

    // Ciklusban beolvassa, tömöríti, majd feldolgozza a tömörített adatokat
    public void Run()
    {
        byte[] inputData;
        compressionStrategy.InitCompression(); // Tömörítés inicializás, impl. függő
        try
        {
            while ((inputData = readData()) != null)
            {
                // Tömörítés, impl. függő
                byte[] compressedData = compressionStrategy.CompressData(inputData);
                processCompressedData(compressedData);
                if (cancellationStrategy.IsCancelled()) // Cancel vizsgálat, impl. függő
                    return;
            }
        }
        finally
        {
            compressionStrategy.CloseCompression(); // Tömörítés lezárás, impl. függő
        }
    }
  …
}
```

Keresztkombinációk (Template Method vs Strategy):

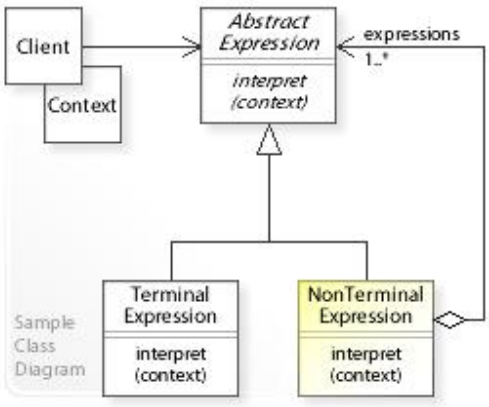


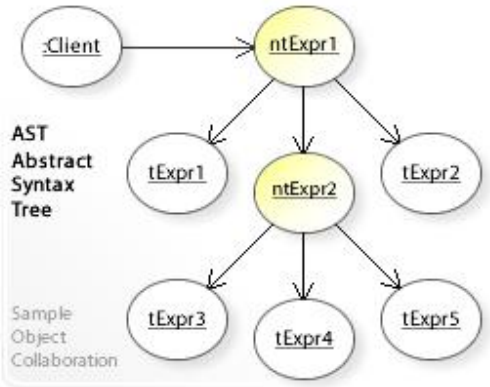
Következmények:

- Új viselkedés könnyen bevezethető, csak egy strategy csere
- Nincs kombinatorikus robbanás keresztkombinációknál
- Unit tesztelhetőséget segíti(dummy strategykkel, amik nem csinálnak semmit, azok működését máskor teszteljük)

Interpreter:

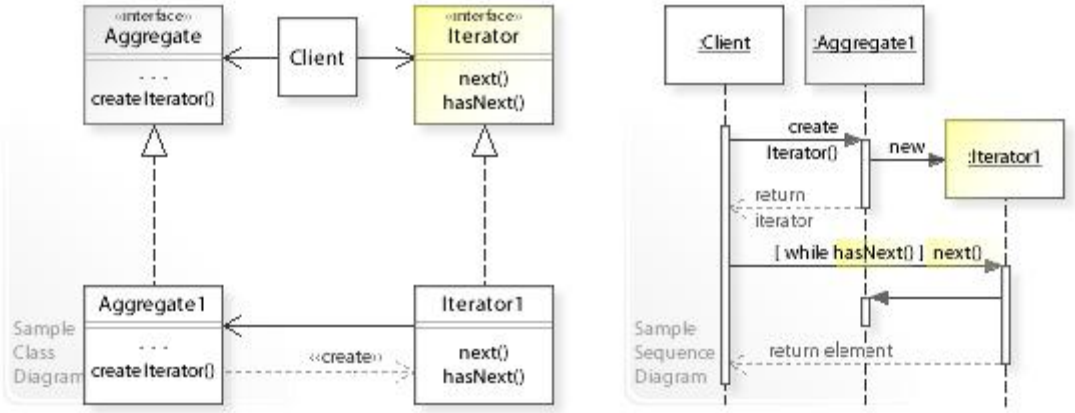
- Egy nyelv mondatainak értelmezéseire szolgaló minta.
- Külön osztályt vezet be a nyelv szimbólumaira(terminálisok, nemterminálisok külön).
- Egyszerű nyelvtanokat értelmez.
- A kifejezéseket egy absztrakt szintaxisfa segítségével értelmezi, ez a fa a kompozit tervezési mintát követi.





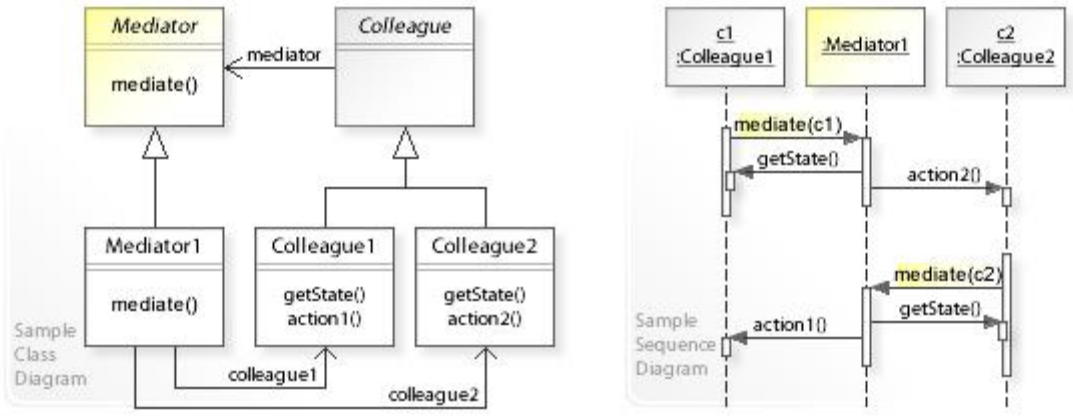
Iterator:

- Tárolók bejárására és elemeik elérésére használt tervezési minta.
- Ez a minta csökkenti az algoritmusok és a tárolók közti csatolást
    - Nem minden algoritmus választható el a tárolótól.
- A tárolók bejárása függetlenedik a belső reprezentációjától.
    - Pl Javában iterátor járja be az összes Iterable interfacet megvalósító objektumot, de teljesen más adatstruktúrákat alkalmazhatnak az osztályok.
- A tároló bejárására találhatunk új módot, annak interfészének változása nélkül.



Mediator:

- Osztályok egymással való interakcióját zárja egységbe.
- Az objektumok nem egymással lépnek interakcióba, hanem egy Mediátor leszármazotton keresztül teszik azt.
- Csökken ezáltal közöttük a csatolás.
- Függetlenül változtatható így az osztályok közt az interakció

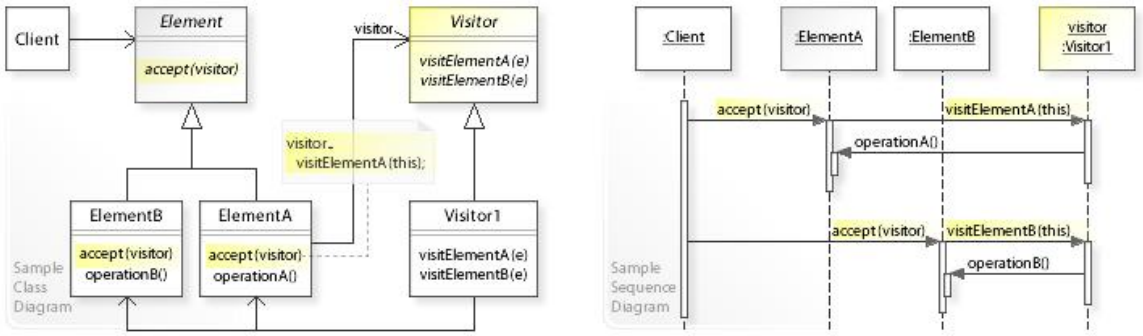


Magyarázat:

- Mediátor adja meg az interfészt a kommunikációra a Colleaguek között
- Mediator1 a Mediator leszármazottja, ez a konkrét implementációja a két Colleague közti interakciónak.
- Colleague1 és 2 ezen a mediátor leszármazotton keresztül kommunikál.

Visitor:

- Ez a minta arra szolgál, hogy az algoritmust különválassza az objektumstruktúrától, amin dolgozik.
- Így könnyebb új műveleteket hozzáadni az objektumstruktúrához, mivel azokat nem kell módosítani.
    - Emiatt a minta követi az open/closed principlet
        - Viszont egyben meg is sérti azt, mert minden új Element(látogatható osztály) hozzáadásakor a Visitorokba bele kell nyúlni.



Acyclic visitor:

- A visitor mintával egy probléma, hogy körkörös függőséget tartalmaz.
- Ezt oldja fel ez a minta, de nem feltétlen jobb.

Pros	Cons
Feloldja a körkörös függőséget.	Párhuzamos Visitor osztályhierarchiát kell csinálni minden Visitable osztályhoz új osztállyal.
Nem okoz fordítási hibát egy új visitor létrehozása	

Konkrét példa: https://java-design-patterns.com/patterns/acyclic-visitor/

Null object:

- Arra ad megoldást, hogy ne kelljen folyamatosan nullcheckeket végezi a kódban.
- A nullobject miatt nem kell nullchecket végezni, ahelyett közös őse van az osztállyal, amin nullchecket hajtanánk végre, és a polimorfizmust használja ki.
- A null object a felüldefiniált függvényeiben nem csinál semmit.
    - Nem keletkezik így NullPointerException
- Alternatívájaként néhány programozási nyelv, például a Kotlin vagy a C# támogat beépített nullcheck operátort.

Egyszerű C++ példa:

```cpp
#include <iostream>

class Animal {
 public:
  virtual ~Animal() = default;

  virtual void MakeSound() const = 0;
};

class Dog : public Animal {
 public:
  virtual void MakeSound() const override { std::cout << "woof!" << std::endl; }
};

class NullAnimal : public Animal {
 public:
  virtual void MakeSound() const override {}
};
```

Strukturális minták:

Facade:

Cél:

- Egységes interfészt biztosít egy alrendszer interfészeinek halmazához.
- Magasabb szintű interfészt definiál, melyen keresztül könnyebb az alrendszer használata
- A Facade-n keresztül férünk hozzá az alrendszerhez
    - Pl Oprendszer, üzleti alkalmazások, compilerek

Pros	Cons
Alrendszer felhasználása egyszerűbb, elég a Facadet ismerni, a belső működést nem kell	Ha az alrendszer sok szolgáltatást nyújt, bonyolult lehet a Facade
Alrendszer szabadon változtatható a Facade mögött	

Használjuk, ha:

- Egyszerű interfacet akarunk biztosítani egy komplex rendszerhez.
- Kliens és alrendszerek közti függőség esetén egy Facade elősegíti az alrendszer függetlenségét
- Layerek (rétegelés) esetén

Adapter:

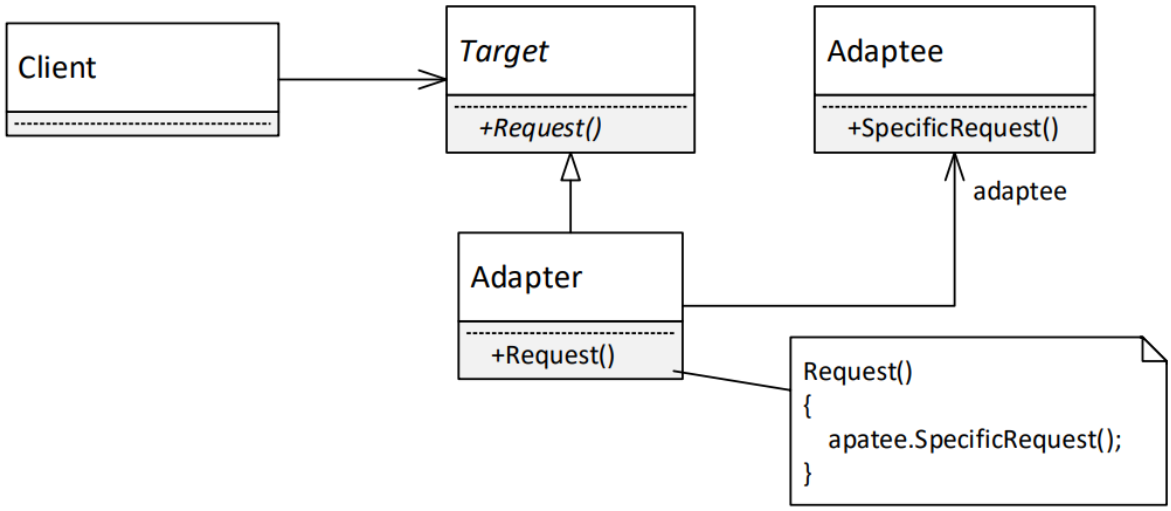
Cél:

- Osztály interfészét olyanná változtatja, amilyet a kliens vár
- Lehetővé teszi egyébként inkompatibilis osztályok együttműködését

Két fajtája van:

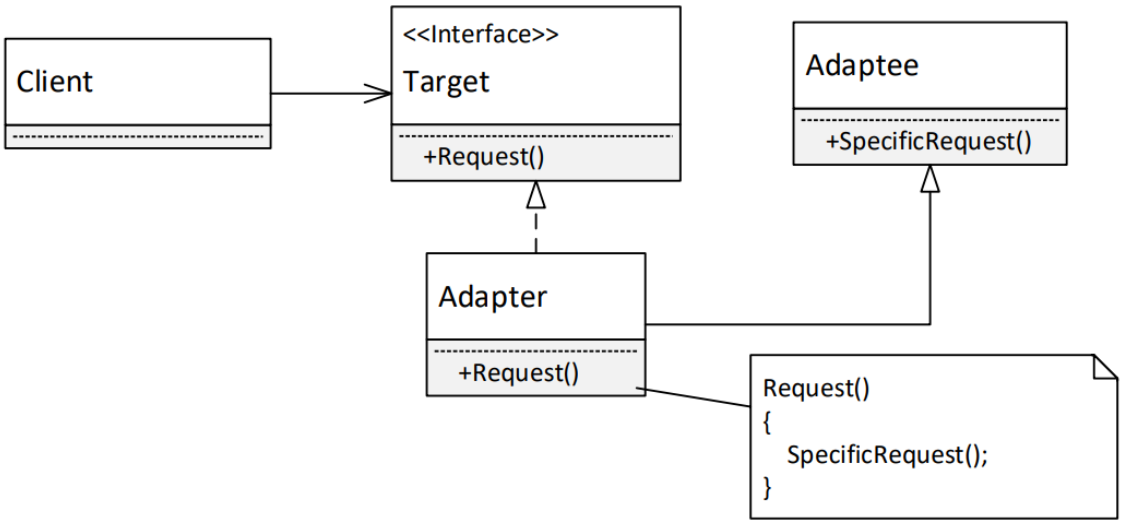
Object Adapter:

Megoldás delegációval



Class Adapter:

Megoldás leszármazással



Magyarázat:

- Adaptee: Az adaptálandó osztály, aminek az interfésze nem megfelelő.
- Adapter: Az illesztő osztály, aminek az interfésze megfelelő, és ahol lehet, a kéréseket az Adaptee segítségével szolgálja ki.
- Target: Az interfész, amit a kliens használ. Lehet interface vagy abstract class is.
    - Javában és C#-ban nincs többszörös öröklés, ezért ott csak interface lehet Class Adapter esetében.
- Általában az Object Adapter a preferált

Akkor használjuk, ha nem tudunk egy osztályt felhasználni azért, mert nem jó az interfésze.

Composite:

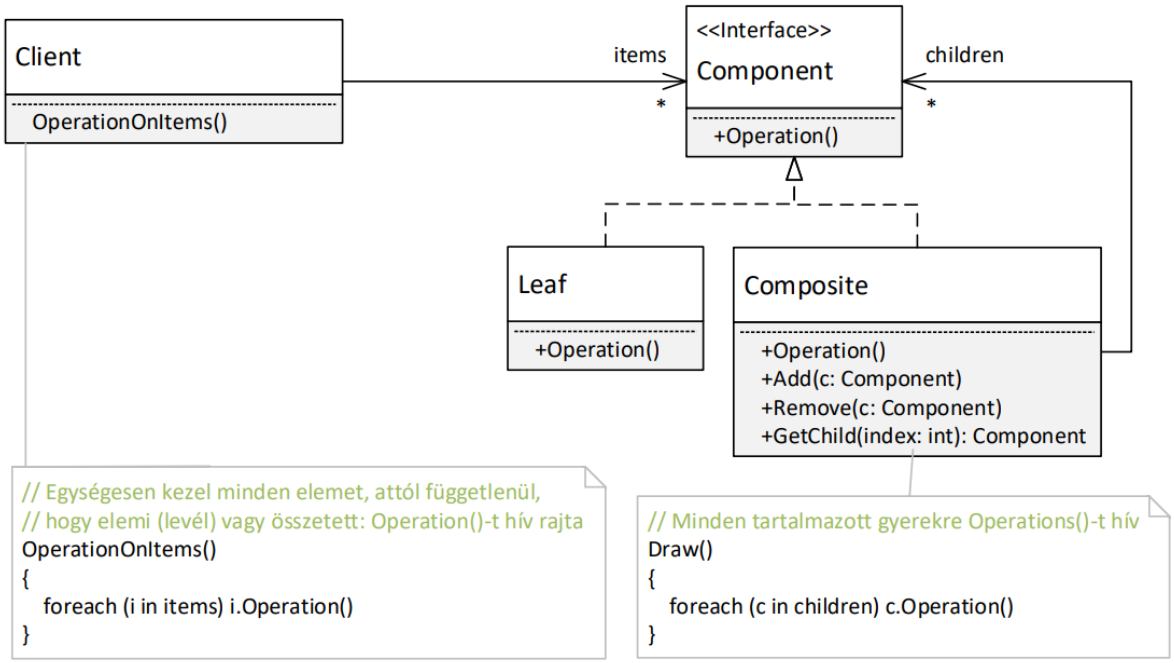
Cél:

- Rész-egész viszonyban lévő objektumokat fastruktórába rendez.
- A kliensek számára lehetővé teszi, hogy az egyszerű és összetett(kompozit) objektumokat egységesen kezelje.

Magyarázat:

- Egységesen kezeli az egyszerű(levél) és összetett objektumokat.

Általánosan:



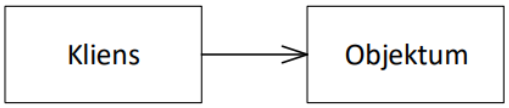
Használjuk, ha objektumok rész-egész viszonyát egységesen akarjuk kezelni, esetleg el is akarjuk rejteni, hogy valami egyszerű, vagy összetett objektum.

Proxy:

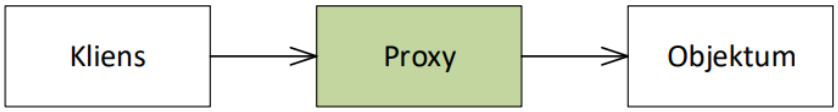
Cél:

- Objektum helyett egy transzparens helyettesítő objektumot használ, mely szabályozza az objektumhoz való hozzáférést.

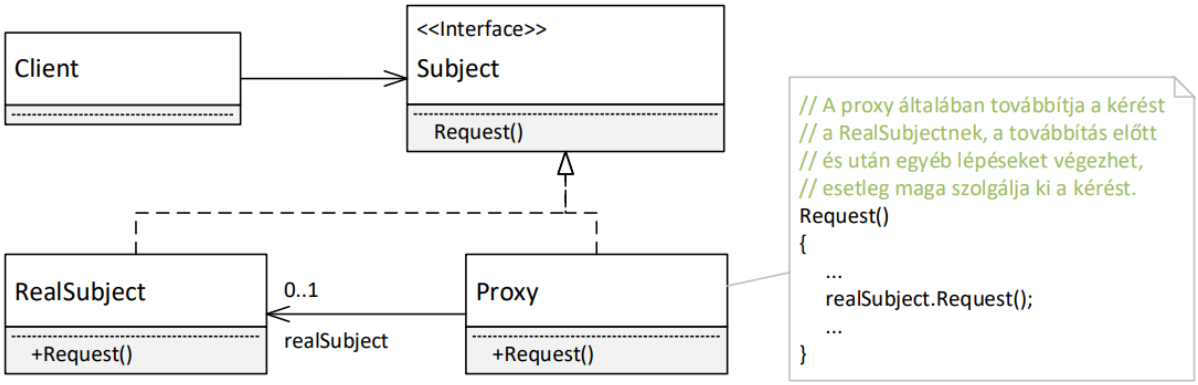
> Itt egy közvetlen kapcsolat van az objektum felhasználója (kliens) és az objektum között



> Itt közbeékeltünk egy proxy-t, mely továbbíthatja a kérést az objektum felé, kiszolgálhatja a kérést az objektum helyett, vagy bármilyen addicionális logikát tartalmazhat



Általánosan:

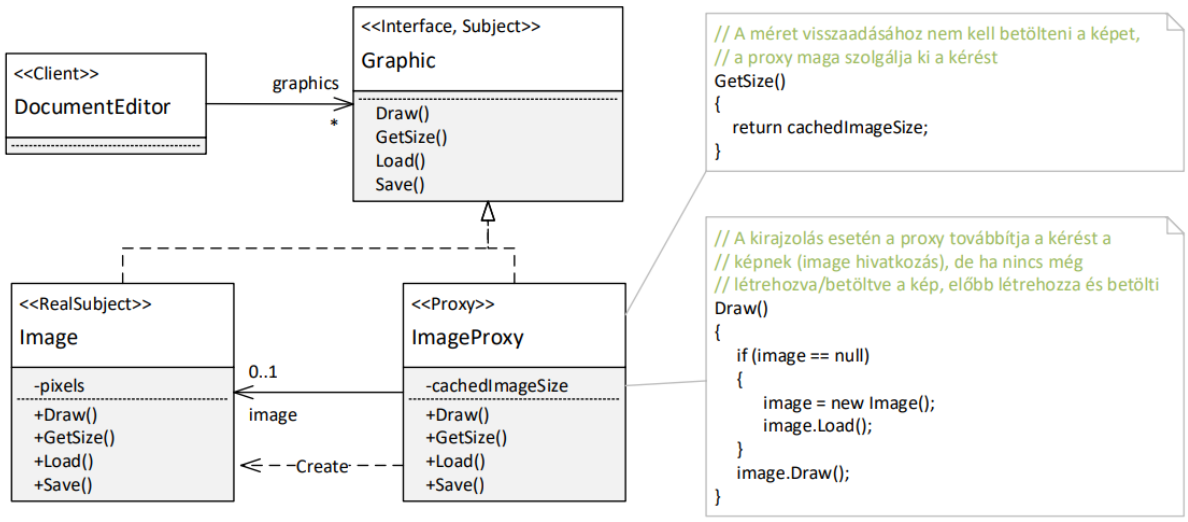


Magyarázat:

- Subject: Közös interfész a Proxy és a valós objektum számára (ezért működik a minta)
- RealSubject: Az elrejtett objektum
- Proxy: Helyettesítő objektum. Pointere(referenciája C#-ban) van a valós objektumra, szabályozza az ahhoz való hozzáférést, akár létrehozhatja és törölheti is azt.

Felhasználása:

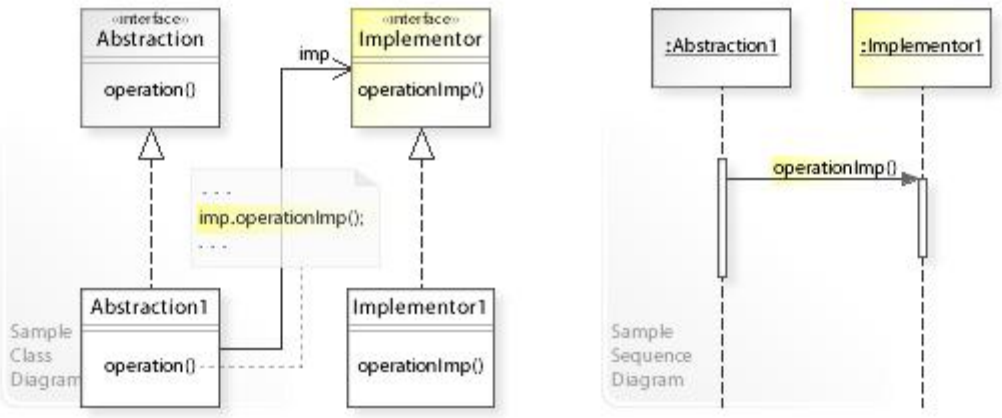
- Virtuális proxy
    - Nagy erőforrásigényű objektumok igény szerinti létrehozása, példa:



- Védelmi proxy
    - Jogok esetén a hozzáférést szabályozza.
- Távoli proxy
    - Távoli objektumok lokális megjelenítése átlátszó módon. A kliens nem érzékeli, hogy a valós objektum máshol van (más címtérben vagy gépen)

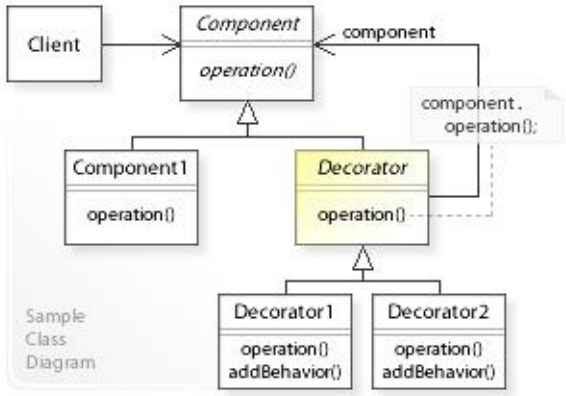
Bridge:

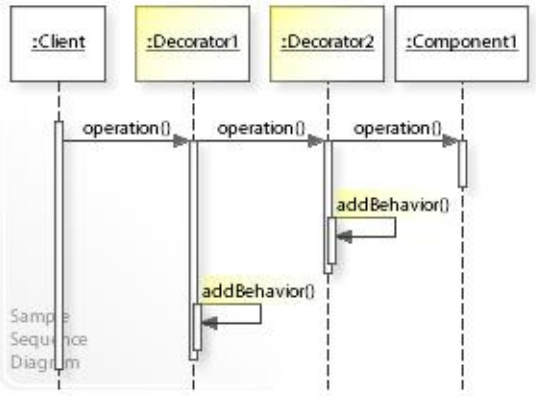
- Ez a minta az absztrakció és az implementáció elválasztására szolgál.
    - Így azok külön változtathatók.
- Az implementációk futási időben cserélhetők, a polimorfizmus miatt.



Decorator:

- A minta arra való, hogy viselkedéssel díszitsünk egy objektumot úgy, hogy az osztály többi objektuma nem változik.



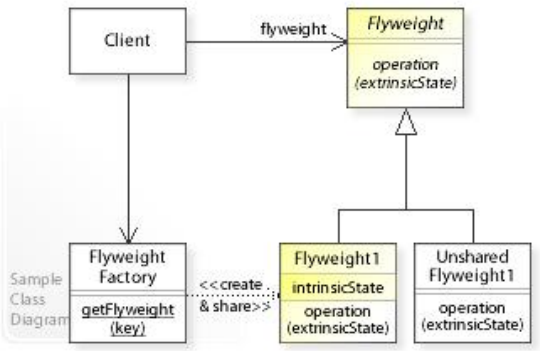


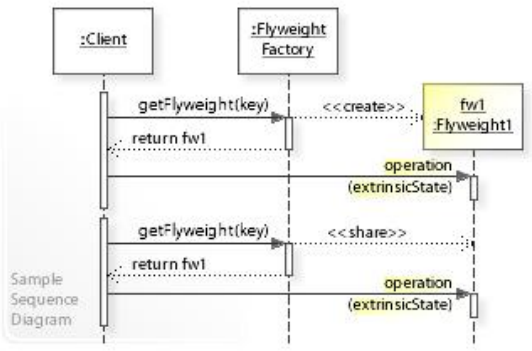
Működés:

- A Decorator konstruktorparaméterként kap egy referenciát egy Componentre.
- A nem változó metódusokat az adapter minta szerint továbbítja az eredeti Componentnek.
- A változó metódusokat felüldefiniálja.

Flyweight:

- Ez a minta a memóriahasználat minimalizálására való.
- Nagyméretű objektumoknál, hogyha van köztük újrahasznosítható adat, ami több objektumnál is ugyanaz, ne tároljuk azokat külön memóriacímen, ezzel pluszban terhelve a memóriát.
- Tároljuk azokat külön külső adatstruktúrákban, és adjuk át az objektumoknak használatkor, hogy spóroljunk a memóriával.





Jegyzetet készítette: Kiss Andor