Szoftvertechnikák

Tervezési minták

Összefoglalás

Készítette:
Siklósi Zsolt
info2007

Bevezetés

Definíció: A **tervezési minta** leír egy gyakran előforduló problémát, annak környezetét és a megoldás magját, amit alkalmazva számos gyakorlati eset hatékonyan megoldható.

Tervezési minták leírása:

- Minta neve (pattern name)
- Probléma és a környezet bemutatása
- Megoldás (solution)
- Következmények (consequences)

Minták csoportosítása hatókör szerint:

1. Architektúrális minták: alapstruktúrát adnak a rendszernek (pl. rétegelés, MVC)
2. Tervezési minták: nyelvfüggetlenek, alrendszereken belül használhatók
3. Idiómák: alacsony szintű, programozási nyelvhez kötődnek (pl. C++ smart pointer)

Minták csoportosítása céljuk alapján:

1. Létrehozási (creational)
2. Strukturális (structural)
3. Viselkedési (behavioral)

Minták csoportosítása érvényesség szerint:

1. Objektum: osztályok kapcsolata, öröklés van a minta középpontjában
2. Osztály: objektumok kapcsolata, asszociáció van a minta középpontjában

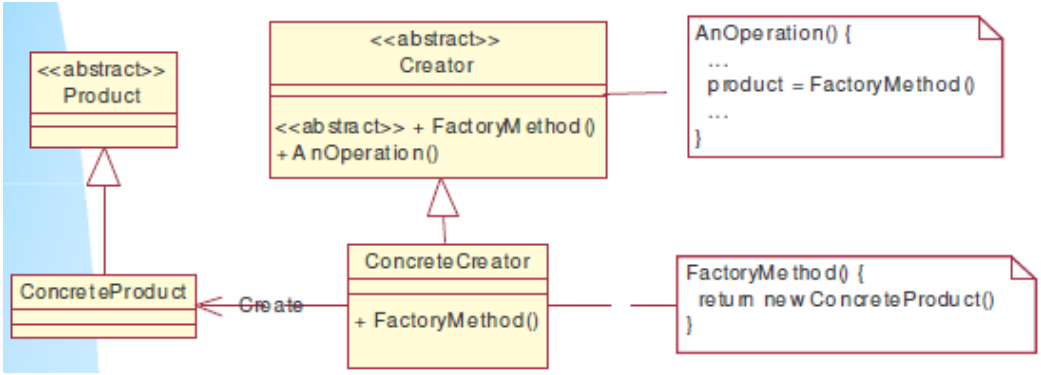
A tervezési minták a tervezési szakaszban segítenek:

- a megfelelő objektumokat megtalálni, definiálni
- a kódot újrafelhasználhatóvá tenni (design for reuse)
- a kódot változtathatóvá, kiterjeszthetővé tenni (design for change)

1. Létrehozási minták

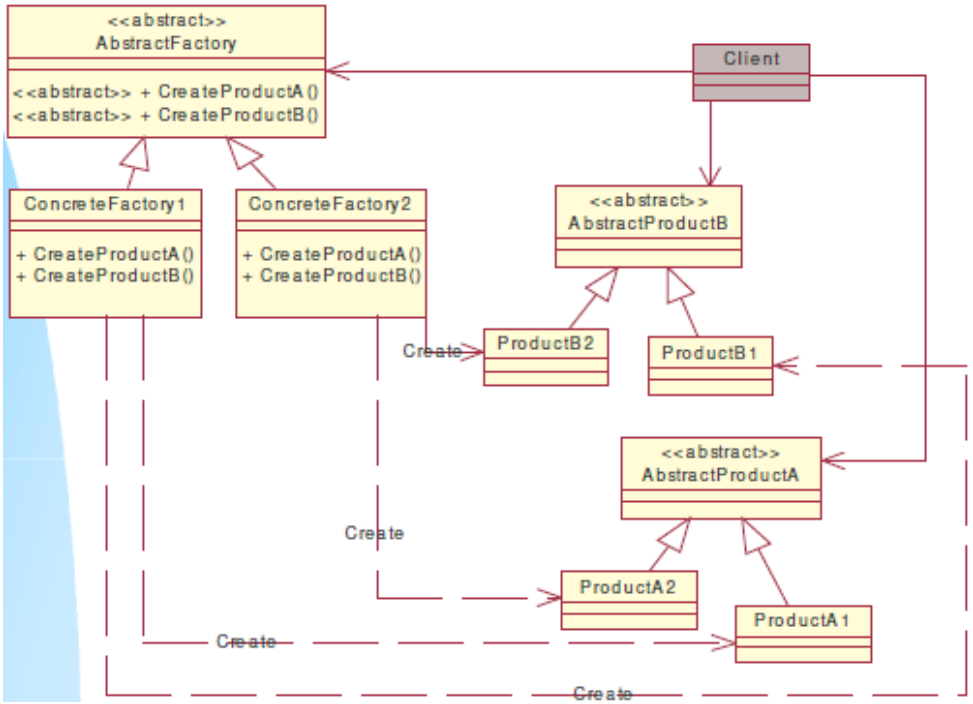
Factory method

Célja: A Factory Method lehetővé teszi, hogy az új példány létrehozását a leszármazott osztályra bízzuk. Interfészt definiál az objektum létrehozására. Szokás virtuális konstruktornak is nevezni.
Példa: Framework, ami egyszerre több dokumentum kezelését támogatja (mint pl. a Visual Studio)
Használjuk, ha egy osztály nem látja előre annak az objektumnak az osztályát, amit létre kell hoznia, vagy ha egy osztály azt szeretné, hogy leszármazottai határozzák meg azt az objektumot, amit létre kell hoznia.
Struktúra:



Abstract factory

Példa: Ablakos rendszerek, GUI vezérlőelemek (ablak, nyomógomb, kiválasztógomb, stb.), több look&feel.
Használjuk, ha a rendszernek függetlennek kell lennie az általa létrehozott dolgoktól ("termék" objektumok, pl. felhasználói felület elemek) vagy a rendszernek több termékcsaláddal kell együttműködnie.
A rendszernek szorosan összetartozó "termék" objektumok adott családjával kell dolgoznia, és ezt akarjuk kényszeríteni a rendszerben.
Előnyök: Elszigeteli a konkrét osztályokat; A termékcsaládokat könnyű kicserélni; Elősegíti a termékek közötti konzisztenciát
Hátrányok: Nehéz új termék hozzáadása. Ekkor az Abstract Factory egész hierarchiáját módosítani kell, mert az interfész rögzíti a létrehozható termékeket (megj.:ezt bizonyos esetekben ki lehet kerülni)
Struktúra:



Singleton

Célja: Biztosítja, hogy egy osztályból csak egy példányt lehessen létrehozni, és ehhez az egy példányhoz globális hozzáférést biztosít.
Megoldás:
Legyen az osztály felelőssége, hogy csak egy példányt lehessen belőle létrehozni
Biztosítson globális hozzáférést ehhez az egy példányhoz.

Az Instance osztály-művelet (statikus) meghívásával lehet példányt létrehozni, illetve az "egyetlen" példányt elérni. C# esetén propertyvel célszerű: Singleton.Instance < globális hozzáférés a példányhoz
A Singleton konstruktora protected láthatóságú! Ez garantálja, hogy csak a statikus Instance metódushíváson keresztül lehessen példányt létrehozni.

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

Használata:

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

Prototype

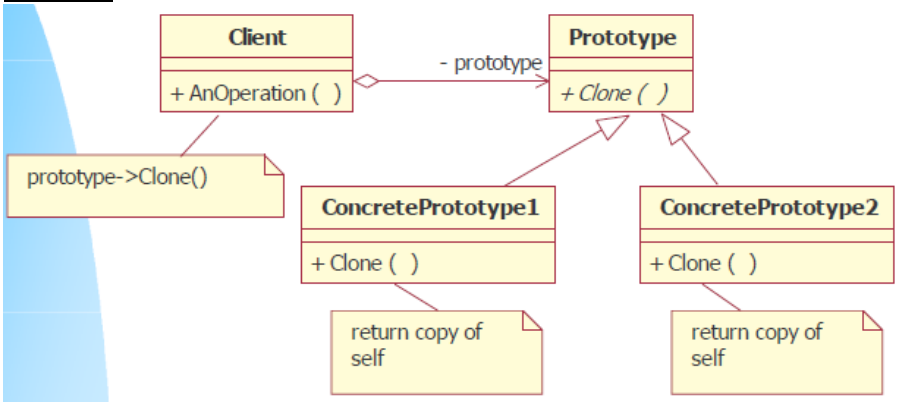
Célja: a prototípus alapján új objektumpéldányok készítése.
Példa: Grafikus keretrendszerre építve kottaszerkesztő alkalmazás elkészítése.
Használjuk, ha egy rendszernek függetlennek kell lennie a létrehozandó objektumok típusától és

- ha a példányosítandó osztályok futási időben határozhatók meg
- ha nem akarunk nagy párhuzamos osztályhierarchiákat
- amikor az objektumok felparam. körülményes, és könnyebb egy prototípust inicializálni majd azt másolni

Minden objektum támogatja (Object osztály művelete): protected Object MemberwiseClone() művelet (shallow copy)
Igazi, mély másolatot végző klónozáshoz implementálható az ICloneable interfész: Object Clone() művelet (deep copy)
Előnyök:

- Objektumok hozzáadása és elvétele futási időben
- Új, változó struktúrájú objektumok létrehozása
- Redukált származtatás, kevesebb alosztály

Hátrányok:
Minden egyes prototípusnak implementálnia kell a Clone() függvényt, ami igen bonyolult lehet
Struktúra:

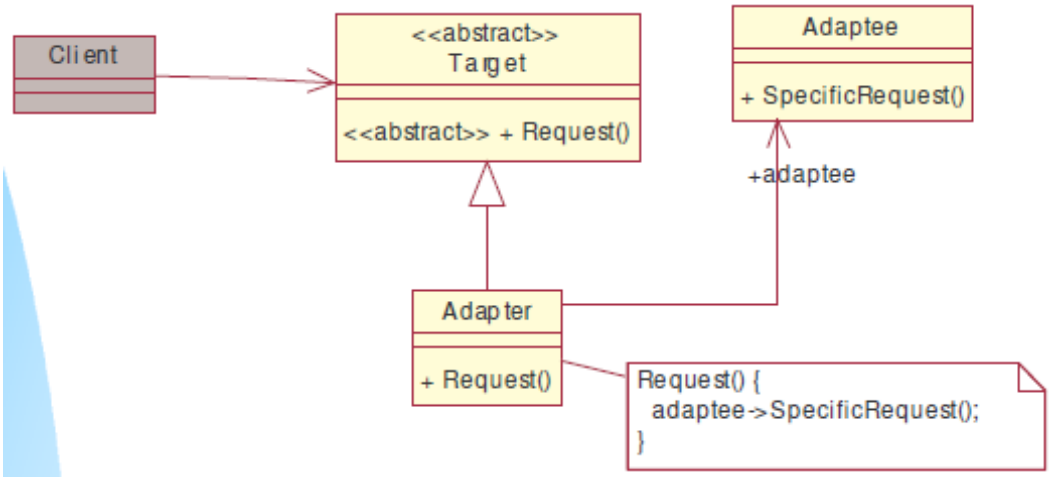


2. Struktúrális minták

Adapter (wrapper)

Célja: 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
Két struktúra:

1. *Class Adapter*: többszörös öröklléssel oldja meg az adaptálást
2. *Object Adapter*: objektum kompozícióval, delegálással oldja meg az adaptálást



(Egy adapter képes több Adaptee-t is magában foglalni, beleértve azok alosztályait is.)

Használjuk, ha

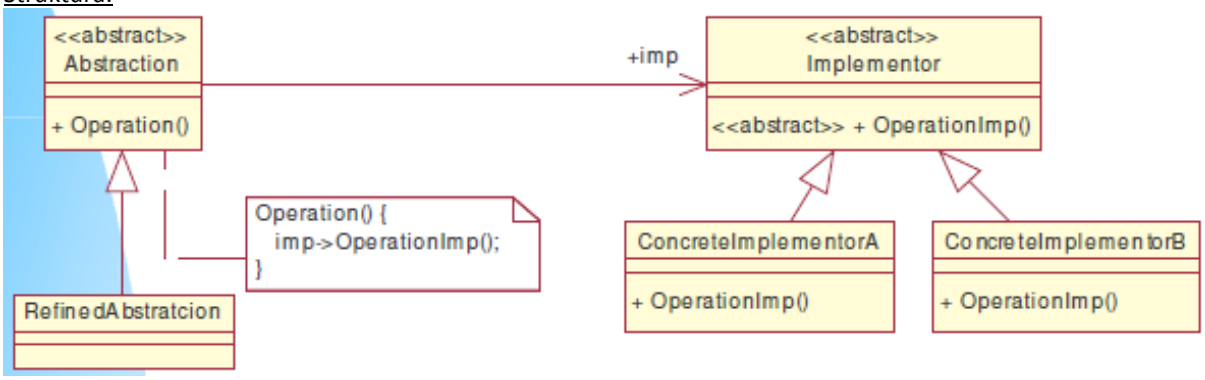
- egy olyan osztályt szeretnénk használni, amelynek interfésze nem megfelelő
- egy újrafelhasználható osztályt szeretnénk készíteni, amely együttműködik előre nem látható vagy független szerkezetű osztályokkal (pluggable adapters)

Bridge

Cél: Különválasztja az absztrakciót (interfészt) az implementációtól, hogy egymástól függetlenül lehessen őket változtatni.
Példa: hordozható ablakozós rendszer XWindow és Presentation Manager alá
Előnyei:

- Az absztrakció és az implementáció különválasztása
- Az implementáció dinamikusan, akár futási időben is megváltoztatható
- Az implementációs részletek a klienstől teljesen elrejthetők
- Az implementációs hierarchia külön lefordított komponensbe tehető, így ha ez ritkán változik, nagy projektek esetén nagymértékben gyorsítható a fordítás/buildelés ideje
- Ugyanaz az implementációs objektum több helyen is felhasználható

Struktúra:

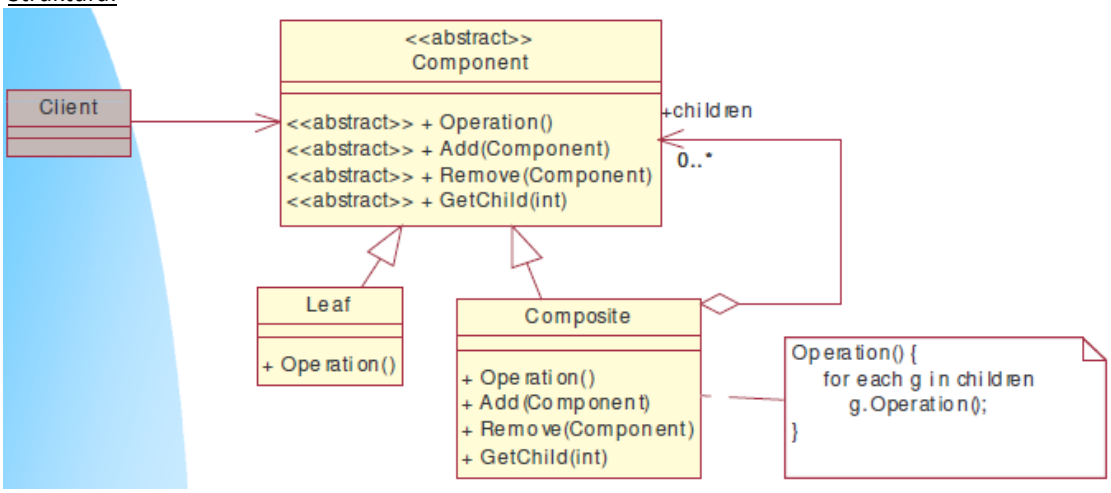


Composite

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
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.

Struktúra:



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), stb. Ezeket tetszőleges kombinációban szeretnénk az osztályokhoz rendelni.
Használjuk, ha

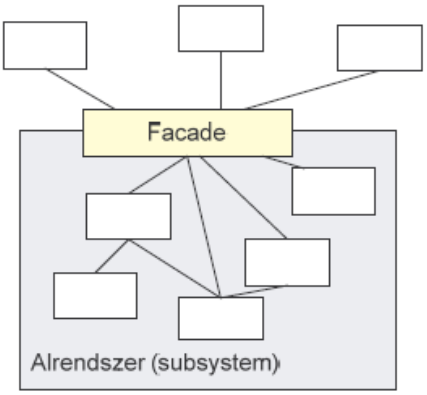
- Dinamikusan szeretnénk funkcionalitást/viselkedést hozzárendelni az egyes objektumokhoz
- A funkcionalitást a kliens számára átlátszó módon szeretnénk az objektumhoz rendelni
- Amikor a származtatás nem praktikus

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.



Facade

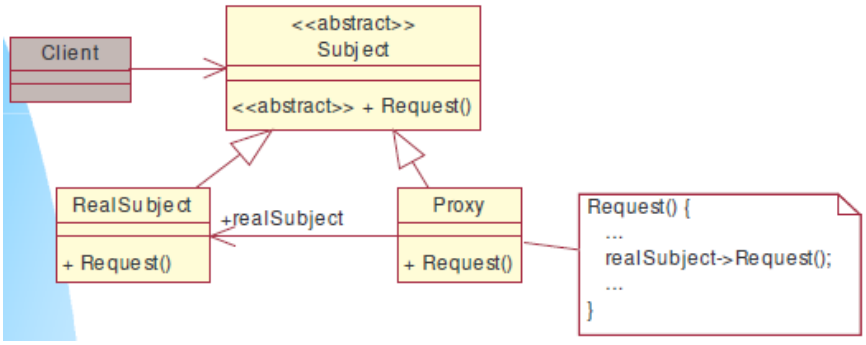
Célja: Egységes interfészt definiál egy alrendszer interfészeinek halmazához. Magasabb szintű interfészt definiál, amin keresztül könnyebb az alrendszer használata.
Példa: Compiler; Többrétegű (többnyire üzleti) alkalmazások

Használjuk, ha

- Egyszerű interfészt szeretnénk biztosítani egy komplex rendszer felé
- Számos függőség van a kliens és az alrendszerek osztályai között. Ilyenkor létrehozva egy Facade-ot elősegíti az alrendszer függetlenségét és a hordozhatóságot
- Layers (rétegelés) esetén

Proxy

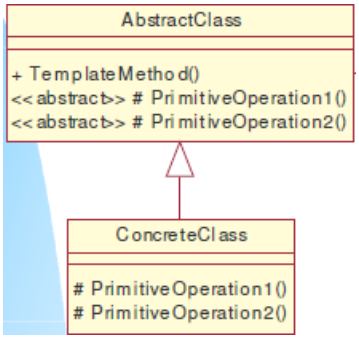
Célja: Objektum helyett egy helyettesítő objektumot használ, ami szabályozza az objektumhoz való hozzáférést
Példa: Szövegszerkesztő (sok nagy méretű kép, nem kell mindet egyszerre megjeleníteni)
Struktúra:



Subject: közös interfészt biztosít a RealSubject és a Proxy számára (ezáltal tud a minta működni)
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 obj. létrehozása és törlése is.
Típusok:
Távoli Proxy: Távoli objektumok lokális megjelenítése átlátszó módon
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

3. Viselkedési minták

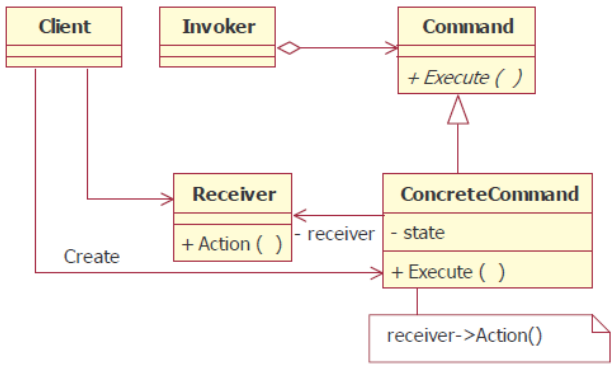
Template method



Célja: Egy műveleten belül algoritmus vázat definiál, és ennek néhány lépésének implementálását a leszármazott osztályra bízza.
Példa: Framework-ben dokumentum megnyitása
Előnyök:
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, melyeket a leszármazott osztályban felül kell/lehet definiálni.

Command (action)

Célja: Egy kérés objektumként való egységbezárása. Ez lehetővé teszi a kliens különböző kérésekkel való felparaméterezését, a kérések sorba állítását, naplózását és visszavonását (undo)
Nagyon rendszerfüggő a koncepció és az implementáció is
Példa: felhasználói parancsok (külön Command leszármazott objektumba zárjuk a kéréseket, és a menüelemeket ezzel paraméterezzük fel)

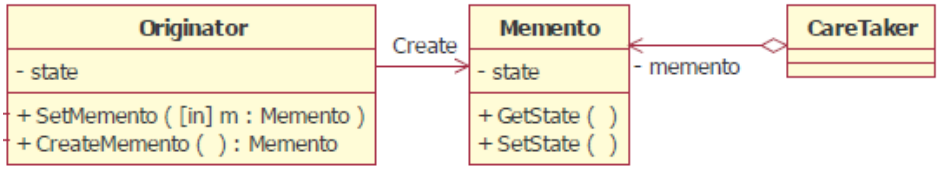


Használjuk, ha
- Strukturált programban callback függvényt használnánk, OO-ban használjunk commandot helyette.
- Szeretnénk a kéréseket különböző időben kiszolgálni. Ilyenkor várakozási sort használunk, a command-ban tároljuk a paramétereket, majd akár különböző folyamatokból/szálakból is feldolgozhatjuk őket.
- Visszavonás támogatására – eltároljuk az előző állapotot a command-ban

Előnyök:
- 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.

Memento

Célja: 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
Struktúra:

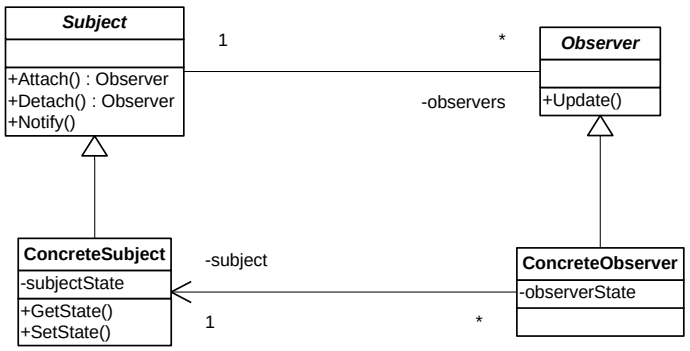


Originator: az ő állapotát kell tudni visszaállítani.
A CreateMemento() elment (pontosabban visszaadja a state állapotot egy Memento objektum formájában)
A SetMemento() visszaállít (pontosabban beállítja a state állapotot a paraméterben megkapott Memento objektum alapján)
Memento: az Originator állapotát tárolja és elméletileg csak az Originator számára biztosít hozzáférést az állapothoz (state).
CareTaker: nyilvántartja a Mementokat
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

Célja: objektumok tudják értesíteni egymást állapotuk megváltozásáról anélkül, hogy függőség lenne a konkrét osztályaiktól
Példa: MVC vagy Document-View architektúra
Lehetővé teszi, hogy egy objektum megváltozása esetén értesíteni tudjon tetszőleges más objektumokat anélkül, hogy bármit is tudna róluk.

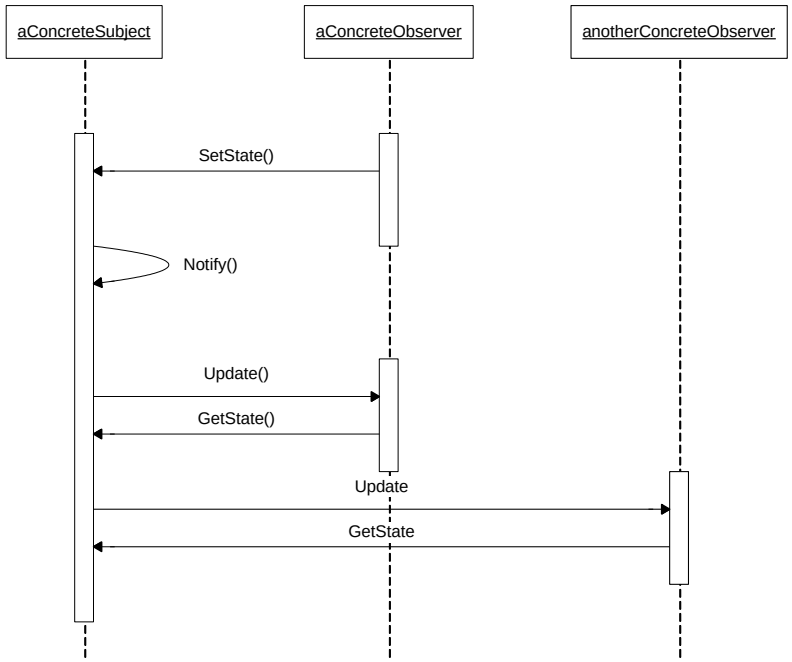


Struktúra:

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; Implementálja 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

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



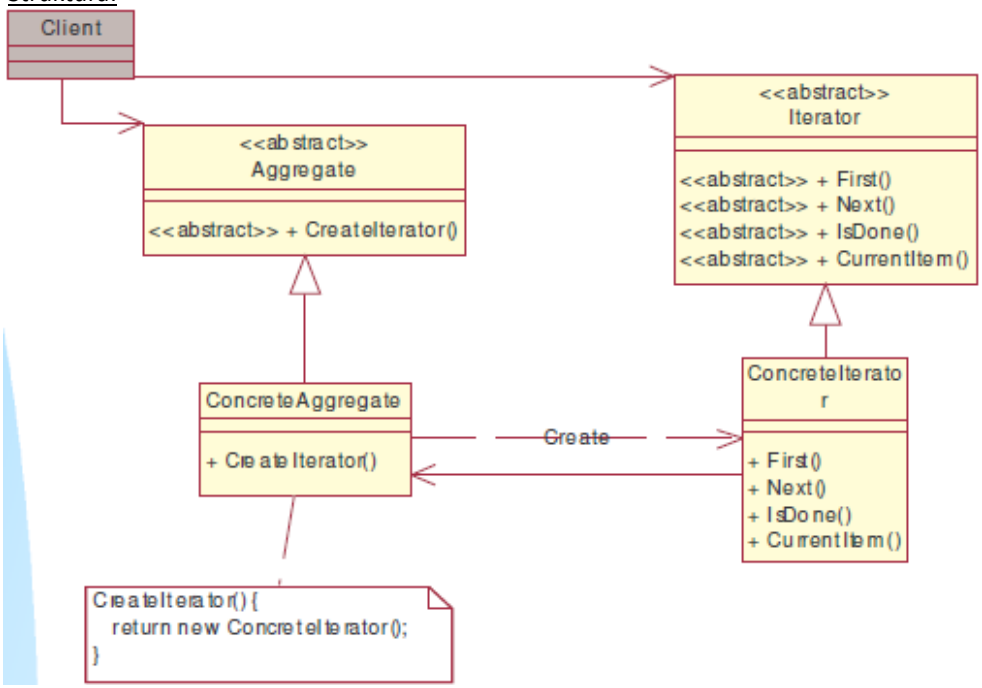
Előnyök: Laza kapcsolat a Subject és az Observer között; Üzenetszórás támogatása
Hátrányok: Felesleges Update-ek; Az egyszerű Update alapján az Subject összes adatát le kell kérdezni

Iterator

Célja: Szekvenciális hozzáférést biztosít egy összetett (pl. lista) objektum elemeihez anélkül, hogy annak belső reprezentációját felfedné
Példa: Lista (hozzáférés a belső struktúra ismerete nélkül)
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

Struktúra:



State

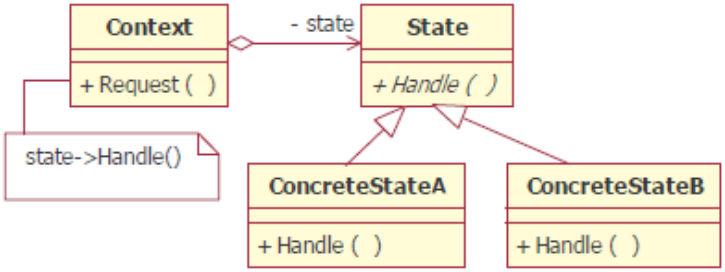
Célja: Lehetővé teszi egy objektum viselkedésének megváltozását, amikor megváltozik az állapota.
Példa: TCPConnection osztály egy hálózati kapcsolatot reprezentál; Három állapota lehet: Listening, Established, Closed; A kéréseket az állapotától függően kezeli
Használjuk, ha

- Az objektum viselkedése függ az állapotától, és a viselkedését az aktuális állapotnak megfelel futás közben meg kell változtatnia
- A műveleteknek nagy feltételes ágai vannak, melyek az objektum állapotától függenek

Előnyök:

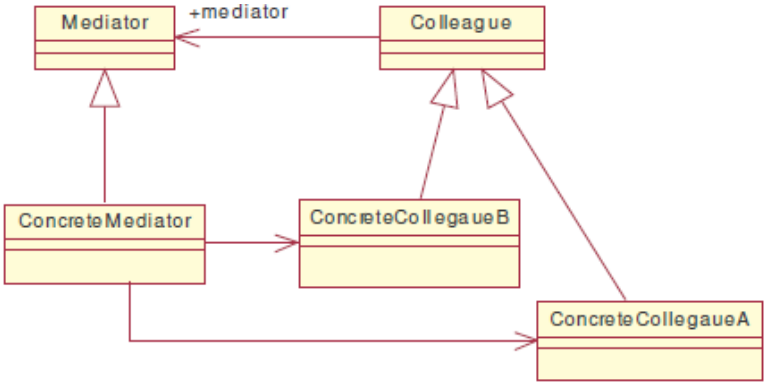
- Egységbezárja az állapotfüggő viselkedést, így könnyű új állapotok bevezetése
- Áttekinthetőbb kód (nincs nagy switch-case szerkezet)
- A State objektumokat meg lehet osztani

Hátrányok: Nő az osztályok száma (csak indokolt esetben használjuk)
Struktúra:



Mediator

Célja: 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 Mediator, a rajtuk levő vezérlőelemek Colleague-ek
Struktúra:
Minden vezérlőelem tartalmaz egy referenciát a dialógus ablakra (mediátor).
A dialógus ablak minden vezérlőelemre tartalmaz egy referenciát.

Strategy

Célja: Algoritmusok egy csoportján belül az egyes algoritmusok egységbe zárása és egymással kicserélhetővé tétele. A kliens szemszögéből az általa használt algoritmusok szabadon kicserélhetők.
Példa: Egy kliens objektum különféle sorrendező algoritmusokat használ
Struktúra: