Gang Of Four (GoF) minták

GoF névvel egy könyv négy szerzőjét szokták illetni, akik a könyvben a szoftvertervezés jellemző problémáira adnak ismétlődő válaszokat. A könyvük első felében az objektumorientált programozás lehetőségeit és csapdáit veszik át, a második részben a 23 klasszikus szoftver tervezési mintát.
Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

- Jellemzően C++ és Smalltalk példákat használnak. Létrehozó: (creational)
  - A minták általában a megadott osztályok további kidolgozásával és példányosítással, illetve példányosításra használhatók…
- Szerkezeti: (structural)
  - A minták objektumösszetételekre épülnek
- Viselkedési (behavioral):
  - A minták a megadott osztályok példányai közötti kommunikációra épülnek

Viselkedési minták: az objektumok közötti kommunikációt leíró minták. Néhány példa:

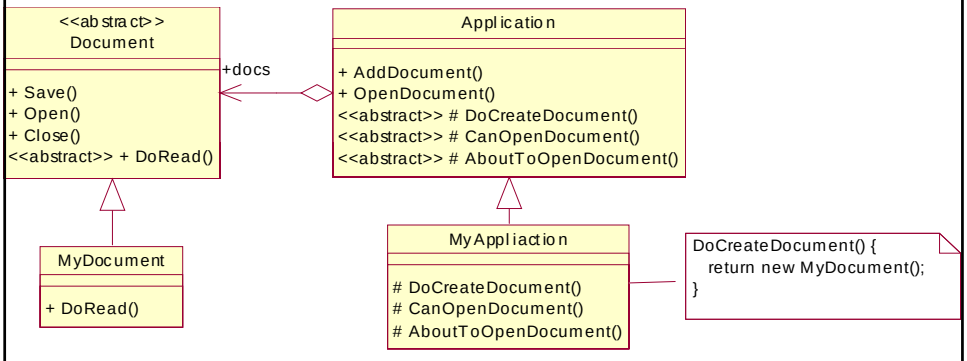
- o Command: objektumba „csomagol” egy kérést a paramétereivel együtt.
- o Iterator: lehetőséget ad egy összetett objektum egyes elemeinek szekvenciális elérésére
- anélkül, hogy a belső struktúrával foglalkozna.
- o Mediator: egy objektumot definiál, amibe „becsomagolja”, hogy más objektumok milyen
- módon működnek együtt. Egy felületet biztosít más felületeknek egy alrendszerben.
- o State: egy módja annak, ahogy egy objektum futásidőben képes legyen részben
- megváltoztatni a saját típusát.

GoF viselkedési minták

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- TemplateMethod
- Visitor

Template Method

- Cél
  - Egy műveleten belül algoritmus vázat definiál, és a váz 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 osztályt leszármaztatnia, amikben megvalósítja az alkalmazás specifikus 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

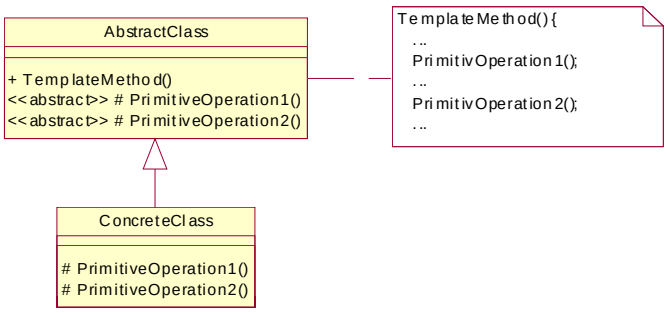
```
// Az Application osztály a framework része

public class Application {
   public void OpenDocument (string 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 != null) {
                _docs.AddDocument(doc);
               AboutToOpenDocument(doc);
                doc.Open();
                doc.DoRead();
        }
}
```

Template method

- Struktúra

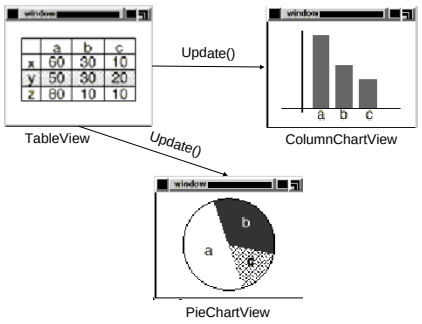


Template method

- Következmények
  - Lehetővé teszi ún. hook függvények definiálását: kiterjesztési pontok
  - Lehetővé teszi, hogy az algoritmus invariáns részeit egy helyen definiáljuk, és a változó részeket a leszármazott osztályban adjuk meg
  - Kód duplikálás 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.
- Megjegyzés
  - Framework-ok esetében gyakori (pl. MFC)

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?



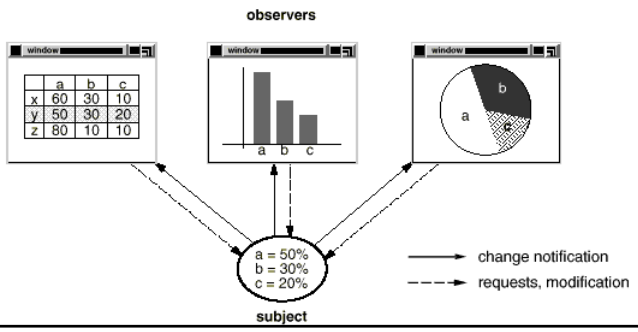
Observer

- 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

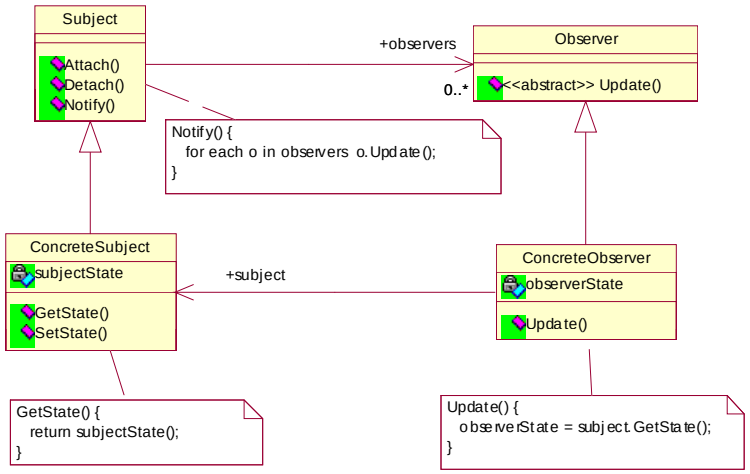


Observer

- **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* pattern

Observer

A klasszikus megvalósítás

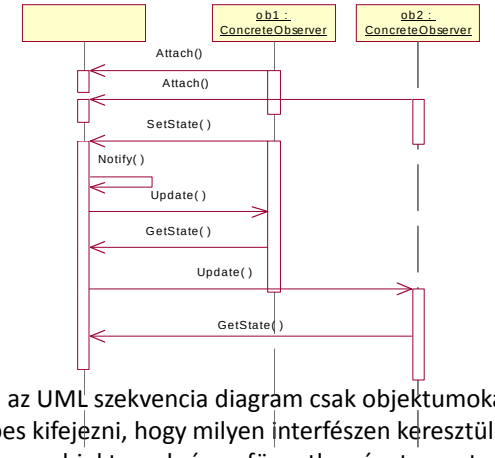


Observer

- Szereplők a klasszikus megvalósításban
  - 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

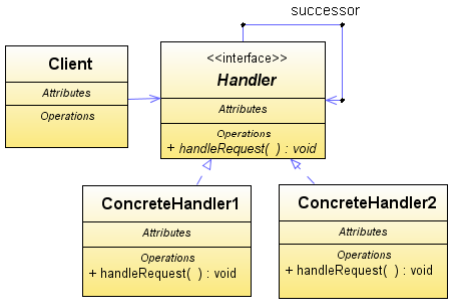
- **Dinamikus viselkedés (az értesítés)**



- „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

Felelősségi lánc

- A kérés küldője és fogadója csak közvetett kapcsolatban van úgy, hogy egy sor, különböző típusú fogadóobjektum közül egy láncon megkeresi azt, amelyik kezeli a kérést

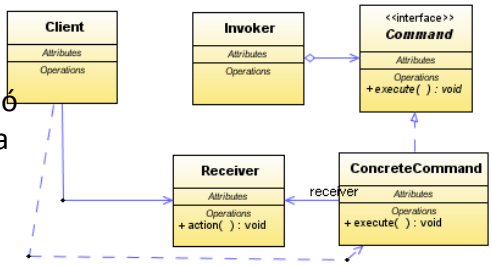


- Client:a kérés kezdeményezője
- Handler felület: definiálja: 1. a kérés kezelőjét 2. a „successor” rákövetkező kapcsolatot
- ConcreteHandlerN: megvalósítja a kéréskezelő eljárást, vagy továbbítja a következőnek

```
class ConcreteHandler1 : Handler
  {
    public override void ProcessRequest(Purchase purchase)
    {
      if (purchase.Amount < 25000.0)
      {
        Console.WriteLine("{0} approved request# {1}",
          this.GetType().Name, purchase.Number);
      }
      else if (successor != null)
      {
        successor.ProcessRequest(purchase);
      }
    }
  }
```

Parancs (Command)

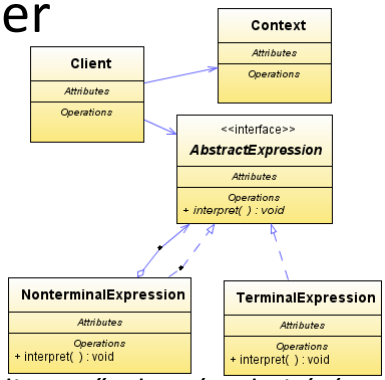
- Kérés becsomagolása objektumként
- Client: létrehoz egy Invoker objektumot, és beállítja a fogadó (Receiver) tagját, majd meghívja egy eljárását.
- Invoker: példányosítja a ConcreteCommand objektumokat és meghívja az execute módszerüket



- Command: a műveletvégrehajtás felületét definiálja
- ConcreteCommand: 1. ő a kapocs a Receiver objektum és a tevékenység között 2. megvalósítja az Execute módszert úgy, hogy meghívja a Receiver egyes módszereit
- Receiver: a parancs egyes műveleteinek megvalósítója

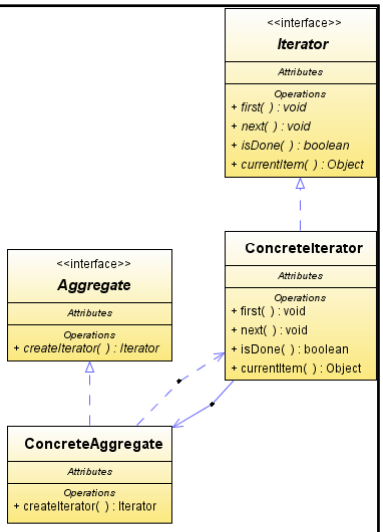
```
class Command {
    public abstract void Execute(); }
class ConcreteCommand implements Command {
    Receiver rec;
    public ConcreteCommand(Receiver rec) {
       this.rec=rec; }
    public override void Execute() {
    …
       rec.action();
    … }
  }
class Receiver {
    public void action(); }
class Invoker {
    private Receiver rec;
    private ArrayList commands = new ArrayList();
    Invoker(Receiver rec){ this.rec=rec; }
    public doIt() {…
    ConcreteCommand cc=new ConcreteCommand(receiver);
    commands.append(cc);
    …
    for each(Command command in commands) {
    command.Execute(); }
    … }
Class Client {
    private receiver=new Receiver();
    private invoker=new Invoker(receiver);
    invoker.doIt();
…
}
```

Interpreter



- Ha adott egy nyelv, akkor megadja a nyelvtan ábrázolását, valamint egy értelmezőprogramot hozzá
- Context: az interpreter számára globális objektumok készlete
- Client: felépíti a program absztrakt szintaxisfáját, beleértve a a TerminalExpression, a NonterminalExpression és más szerkezeteket
- AbstractExpresson: felület, amely megadja a művelet végrehajtását
- NonterminalExpression: – minden nemterminálishoz készítünk egy ilyen osztályt, benne a szabály jobboldalának megfelelő AbstractExpression példányokkal – megvalósítja az interpret műveletet
- TerminalExpression: - megvalósítja az interpret műveletet – a mondat minden terminálisához készítünk egy példányt

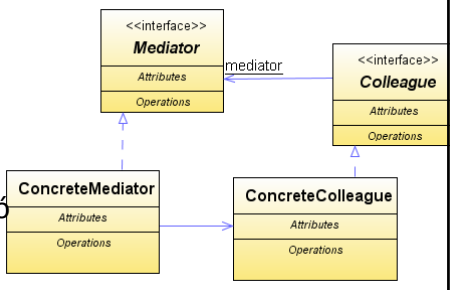
Iterator



- (Lineáris) Gyűjtemény elemeinek szekvenciális elérése
- Iterator: absztrakt felület gyűjteményelemek bejárásához és eléréséhez
- ConcreteIterator: - az Iterátor felület megvalósítása – a gyűjtemény aktuális pozíciójának tárolása
- Aggregate: absztrakt gyűjtemény – iterátor létrehozását szabja meg
- ConcreteAggregate: konkrét iterátor létrehozása

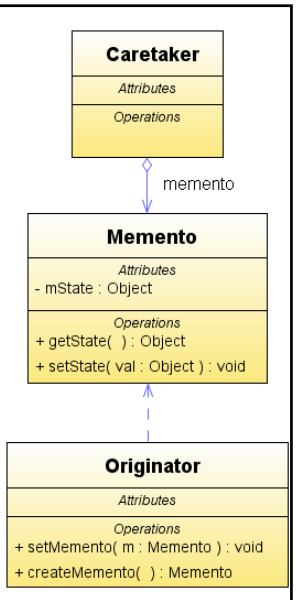
Mediator

- Objektum megadása, amely más objektumok kölcsönhatását zárja egységbe úgy, hogy laza (nem közvetlen) kapcsolatot hoz közöttük létre
- Mediator: a Colleague objektumok kölcsönhatásainak kezelését megadó felület
- ConcreteMediator: - megvalósítja a Mediátor felületét – tárolja a kollégákat
- ConcreteColleague: - rámutat a Mediátorára – kommunikációs igény esetén a másik kolléga helyett a mediátorral kommunikál



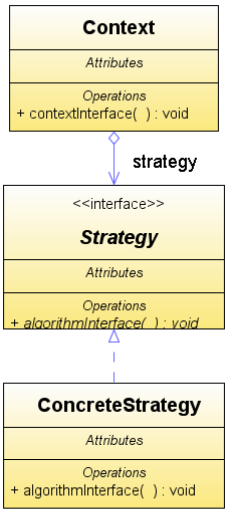
Memento

- Egy objektum belső állapotának mentése később visszatöltéshez
- Memento: - az Originator objektum belső állapotát (v. annak egy részét) tárolja – két felület: 1.Caretaker csak a Mementót látja, és csak hozni-vinni tudja. Csak Originator tud a teljes belső állapothoz hozzáférni
- Originator: - a belső állapotáról készít egy Memento pillanatfelvételt – a Mementó segítségével visszaállítja a belső állapotát
- Caretaker: - tárolja a mementót – de sosem fér hozzá a mementóhoz közvetlenül



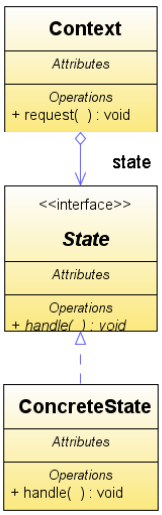
Strategy

- Algoritmuscsalád megadása, amelyek cserélhetők. Strategy lekezeli az algoritmusok változását úgy, hogy az ügyfé nem látja
- Strategy: Felületmegadás, amin keresztül pl. Context meghívja
- ConcreteStrategy: a Strategy felület algoritmusainak megvalósítása
- Context: - a ConcreteStrategy objektumma együtt konfigurálható
  - egy Stratégia objektumra mutató referenciát tárolunk.
- Megadható egy felület is, amin keresztül Strategy hozzáférést biztosíthat



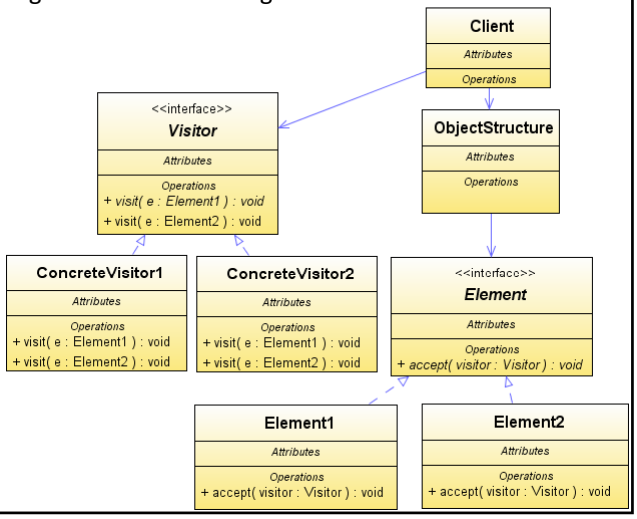
State

- A minta célja, hogy egy objektum a konkrét típusát futásidőben is képes legyen változtatni…
- A konkrét típus a State osztályból van leszármaztatva (esetleg több is) ConcreteState. A Context-beli eljárások csak az absztrakt State felület eljárásait hívják meg.



Visitor

- Visitor kiterjeszti az Element (könnyű) objektumok működését anélkül, hogy az Element osztályhierarchiához hozzá kellene nyúlni.
- Dupla polimorfizmus: 1. Element konkrét típusától függően más és más accept megvalósítás hívódik meg. 2. Az accept-nek átadott Visitor konkrét típusától függően más és más visit megvalósítás hívódik meg…



Visitor

- Client: aki az eszközt használja. Kapcsolódik az ObjectStructure-szerkezetben található Element-ekhez, ill. Visitor osztálypéldányokhoz. Meghívja az Element:accept(Visitor) eljárást.
- Visitor: definiálja a visit eljárásokat minden konkrét Visitorra. A visit eljárások elvégzik a szükséges műveleteket
- Element: definiálja az accept(Visitor) eljárást. Az accept eljárás meghívja a Visitor:visit(this) eljárást, ami a ‘this’ típustól függően meghívja a Visitor:visit(Element) eljárást.