

**Eötvös Loránd Tudományegyetem  
Informatikai Kar**

## **Programozási technológia II**

---

### **6. előadás**

## **Objektumorientált tervezési szempontok és minták**

---

**© 2016 Giachetta Roberto**  
**groberto@inf.elte.hu**  
**<http://people.inf.elte.hu/groberto>**



# Objektumorientált tervezési szempontok és minták

## A tervezés

- Az objektumorientált tervezés során öt alapelvevet célszerű követnünk (*SOLID*):
  - *Single responsibility principle* (SRP)
  - *Open/closed principle* (OCP)
  - *Liskov substitution principle* (LSP)
  - *Interface segregation principle* (ISP)
  - *Dependency inversion principle* (DIP)
- A tervezés során mintákra hagyatkozunk, a szoftver teljes architektúráját definiáló mintákat nevezzük *architekturális mintáknak* (*architectural pattern*), az architektúra alkalmazásának módját, az egyes komponensek összekapcsolását segítik elő a *tervminták* (*design pattern*)

# Objektumorientált tervezési szempontok és minták

## Single responsibility principle

- A *Single responsibility principle (SRP)* kimondja, hogy egy programegység (komponens, osztály, metódus) csak egy felelősséggel rendelkezhet
  - a felelősség az a tárgykör, ami változtatás alapegységeként szolgálhat („*reason for a change*”)
    - műveletek és adatok egy halmaza, amelyet ugyanannak az üzleti szerepkörnek (*business role*) a része, és egy követelmény teljesítését biztosítják
  - azonosítanunk kell a szereplőket, majd a követelményeiket, végül a tárgyköröket összefüggéseik szerint (milyen tényezők változtathatóak egymástól függetlenül), ennek megfelelően alakítjuk ki a felelőségeket

# Objektumorientált tervezési szempontok és minták

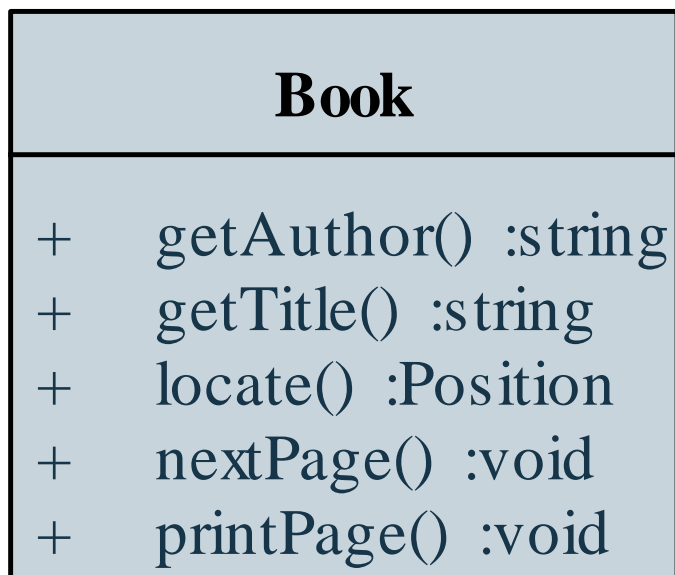
## Single responsibility principle

- számos, jól megkülönböztethető felelősségi kör található az alkalmazásokban:
  - perzisztencia, üzleti logika, vizualizáció, felhasználói interakció
  - adatformázás, konverzió, validáció
  - hibakezelés, eseménynaplózás
  - típuskiválasztás, példányosítás
- elősegíti a programegységek közötti laza csatolást, viszont túlzott használata feltöredezheti a programszerkezetet
  - célszerű csak azokat a változásokat figyelembe venni, amik várhatóak bekövetkezhetnek

# Objektumorientált tervezési szempontok és minták

## Single responsibility principle

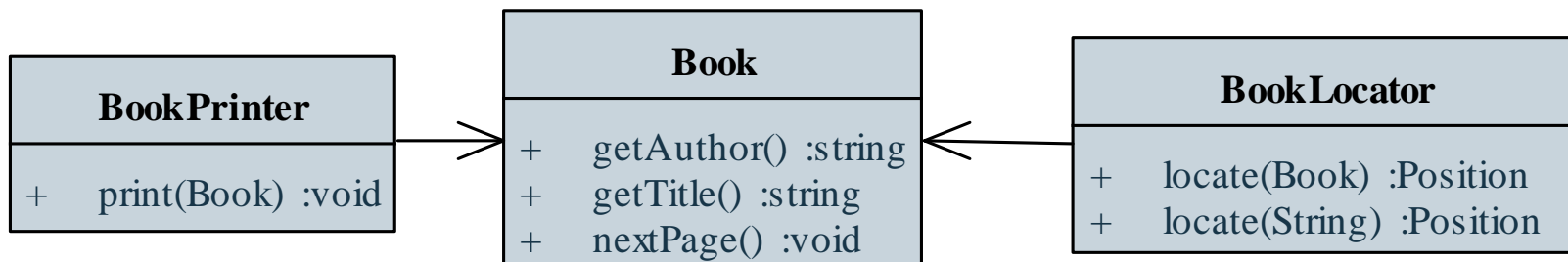
*Példa:* Modellezzünk egy könyvtári könyv (**Book**), amelynek van szerzője (**author**), címe (**title**), lehet lapozni (**nextPage**), kiíratni (**printPage**), és megkeresni a helyét a könyvtárban (**locate**).



# Objektumorientált tervezési szempontok és minták

## Single responsibility principle

- a könyv elérhető két szerepkörben, az olvasó és a könyvtáros számára is
- a kiíratás csak a olvasóra tartozik, és számos módja lehet, ezért célszerű a leválasztása
- a keresés csak a könyvtárosra tartozik, a kölcsönzőre nem, ugyanakkor szintén több módon is elvégezhető, ezért célszerű a leválasztása



# Objektumorientált tervezési szempontok és minták

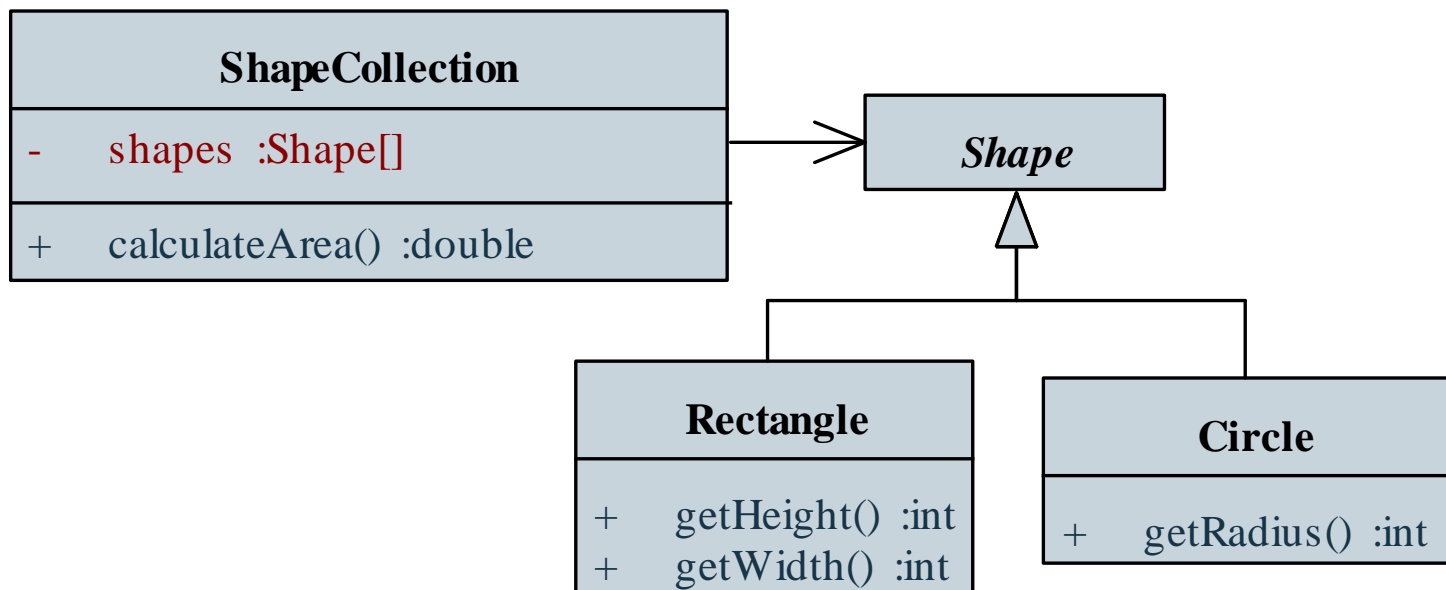
## Open/closed principle

- *Open/closed principle* (OCP) kimondja, hogy a programegységek nyitottak a kiterjesztésre, de zártak a módosításra
  - a programszerkezetet úgy kell megvalósítanunk, hogy új funkciók bevezetése ne a jelenlegi programegységek átírását igényelje, sokkal inkább újabb programegységek hozzáadását
  - a kiterjesztést általában öröklődés segítségével valósítjuk meg
- A SRP és az OCP kiegészítik egymást, mivel az új funkciók egy új felelősség bevezetését jelenti, így általában egyszerre szegjük meg a két elvet

# Objektumorientált tervezési szempontok és minták

## Open/closed principle

*Példa:* Modellezzünk geometriai alakzatokat (**Shape**), speciálisan téglalapot (**Rectangle**) és kört (**Circle**). Az alakzatokat csoportosítsuk gyűjteménybe (**ShapeCollection**), és tegyük lehetővé az összterület kiszámítását (**calculateArea**).





# Objektumorientált tervezési szempontok és minták

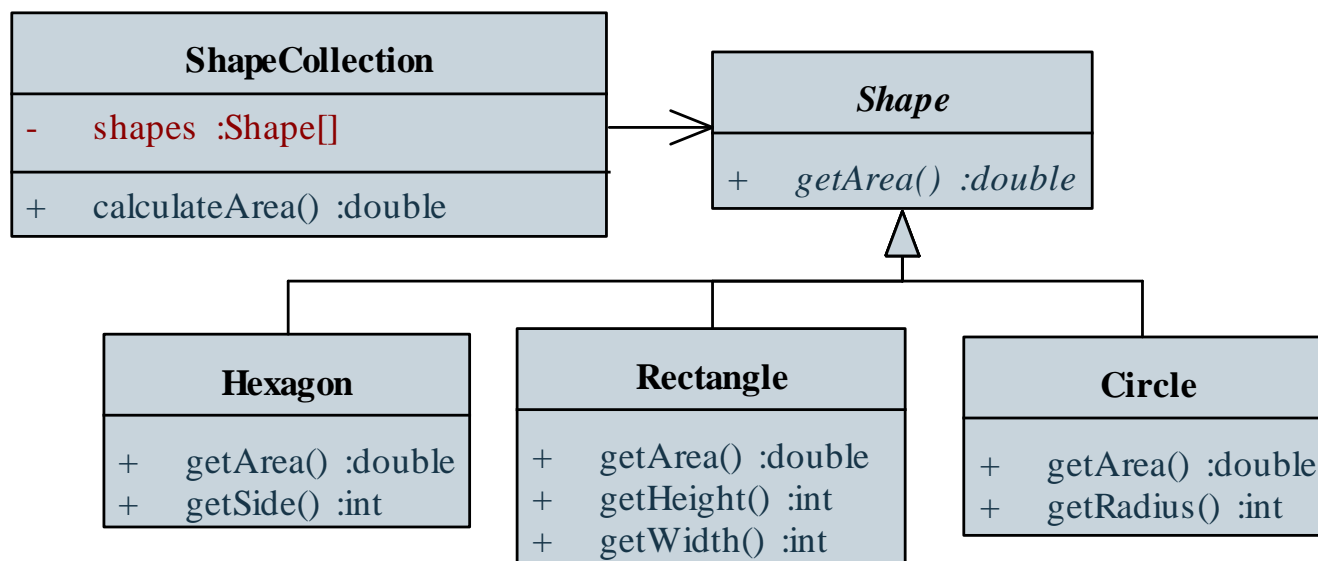
## Open/closed principle

```
double calculateArea() {
    double area = 0;
    foreach (Shape shape : shapes) {
        if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle)shape;
            area += r.width() * r.height();
        }
        if (shape instanceof Circle) {
            Circle c = (Circle)shape;
            area += Math.pow(c.radius(), 2) *
                Math.PI;
        }
    }
    return area;
}
```

# Objektumorientált tervezési szempontok és minták

## Open/closed principle

- egy új alakzat (pl. hatszög) hozzáadása esetén módosítanunk kell a metódust, új ágat bevezetve
- ha minden alakzat ki tudja számítani a saját területét (**area**), akkor az új alakzattal egy új terület képletet lehetne megadni, és nem kellene módosítani a területszámítást



# Objektumorientált tervezési szempontok és minták

## Open/closed principle

```
double calculateArea() {
    double area = 0;
    foreach (Shape shape : shapes) {
        area += shape.getArea();
    }
    return area;
}
```

...

```
class Hexagon implements Shape {
    ...
    public double getArea() {
        return ...;
    }
}
```

# Objektumorientált tervezési szempontok és minták

## Liskov substitution principle

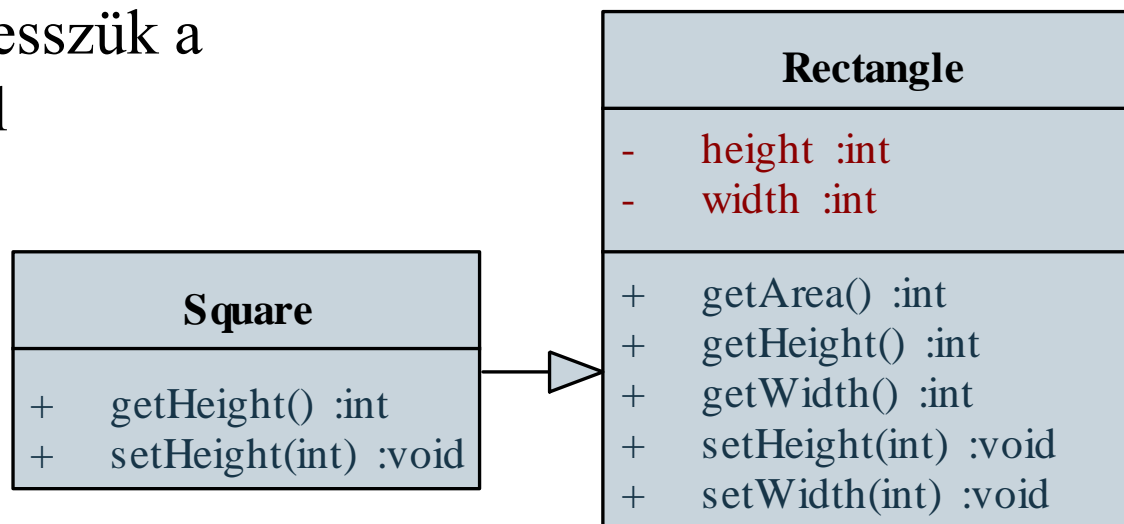
- A *Liskov substitution principle* (LSP) értelmében bármely típus (osztály) példánya helyettesíthető altípusának egy példányával úgy, hogy közben a program tulajdonságai (funkcionális és nem funkcionális követelményei) nem változnak
  - megszorításokat tesz a szignatúrára
    - elvárja a paraméterek kontravarianciáját, a visszatérési értékek kovarianciáját
    - tiltja a kivételek típusának bővítését
  - megszorításokat tesz a viselkedésre
    - elvárja az invariánsok megtartását
    - tiltja az előfeltételek erősítését, az utófeltételek gyengítését

# Objektumorientált tervezési szempontok és minták

## Liskov substitution principle

*Példa:* Modellezzünk téglalap (**Rectangle**) és négyzet (**Square**) alakzatokat, ahol a négyzet egy speciális esete a téglalapnak.

- mindkettőnek megadhatjuk a szélességét/magasságát, és lekérdezhetjük a területét
- a négyzet esetén a magasságot egyenlővé tesszük a szélességgel



# Objektumorientált tervezési szempontok és minták

## Liskov substitution principle

```
class Rectangle { // téglalap
    ...
    public int getHeight() { return height; }
    public int setHeight(int h) { height = h; }
    public int getArea() {
        return getWidth() * getHeight();
    }
}
...
class Square : Rectangle { // négyzet
    public int getHeight() { return getWidth(); }
    public int setHeight(int h) { setWidth(h); }
}
```

# Objektumorientált tervezési szempontok és minták

## Liskov substitution principle

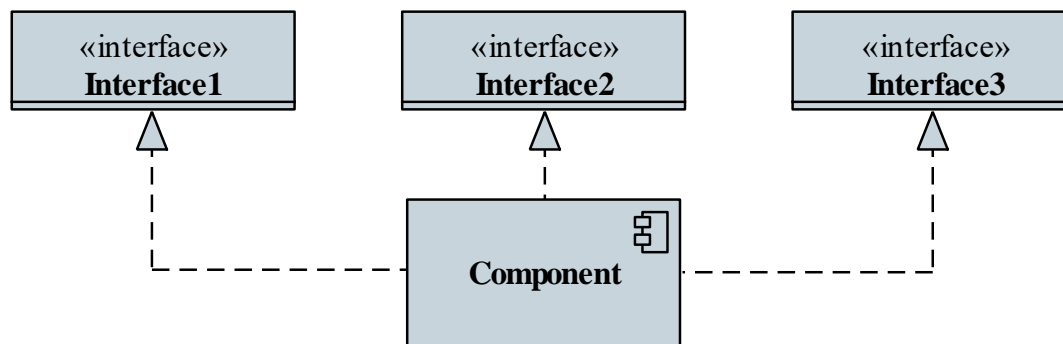
```
Rectangle rec = new Rectangle();
rec.setWidth(4);
rec.setHeight(6);
System.out.println(rec.GetArea()); // 24
    // elvárt viselkedés

rec = new Square();
rec.setWidth(4);
rec.setHeight(6);
    // kicseréljük a példányt a leszármazott
    // típusra
System.out.println(rec.GetArea()); // 36
    // nem az elvártnak megfelelően viselkedik
```

# Objektumorientált tervezési szempontok és minták

## Interface segregation principle

- Az *Interface segregation principle* (ISP) szerint egy kliensnek sem szabad olyan interfésztől függenie, amelynek műveleteit nem használja
  - ezért az összetett interfészeket célszerű több, kliens specifikus interfészre felbontani, így azok csak a szükséges műveleteiket érhetik el
  - a műveletek megvalósításait továbbra is összeköthetjük

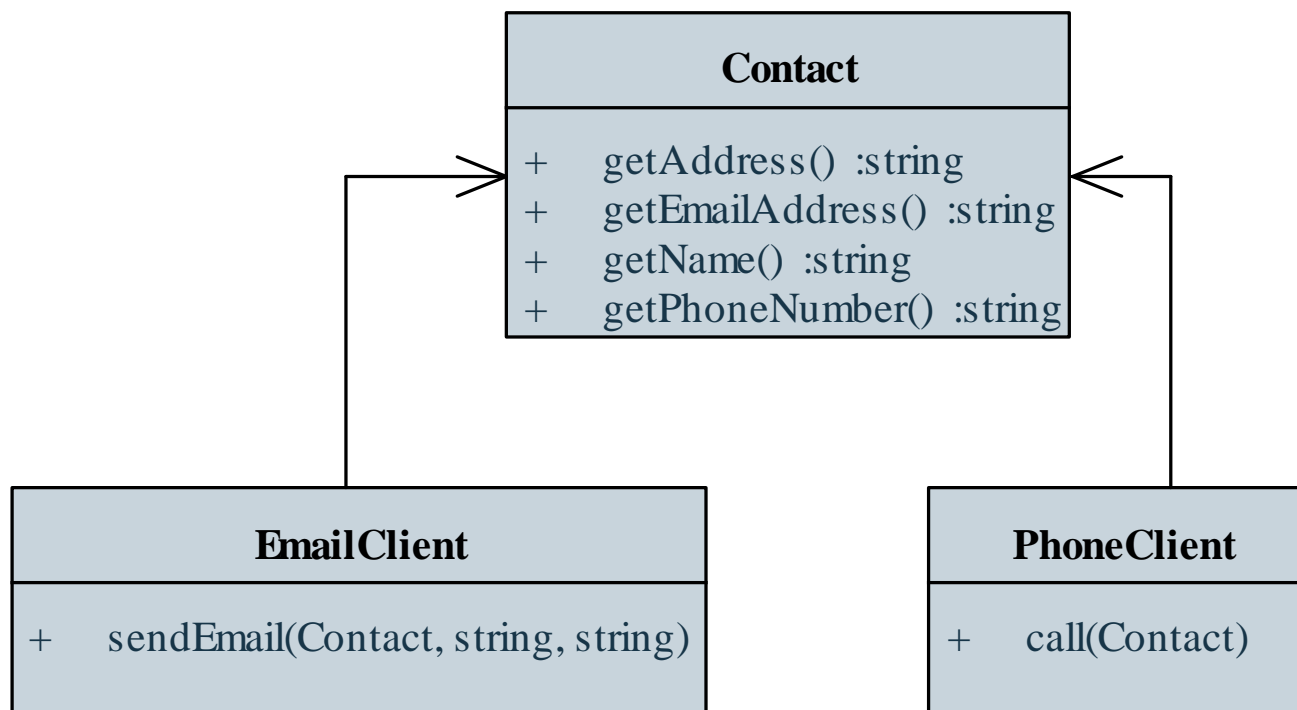




# Objektumorientált tervezési szempontok és minták

## Interface segregation principle

*Példa:* Modellezzünk egy névjegyet (**Contact**), ahol kliensek küldhetnek e-mailt (**EmailClient**), illetve hívhatják (**PhoneClient**) a címzettet az adatok alapján.



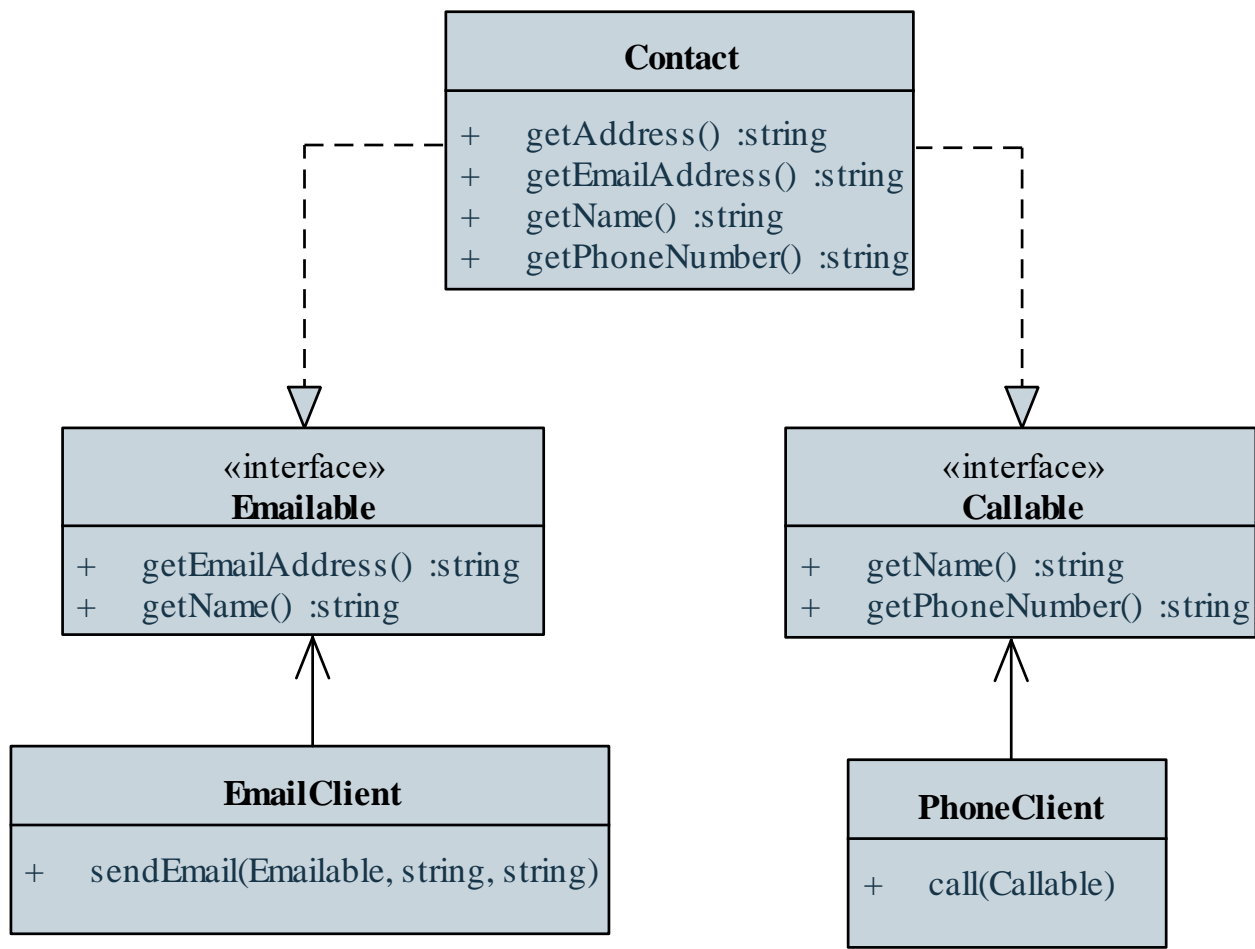
# Objektumorientált tervezési szempontok és minták

## Interface segregation principle

- mindkét kliens hozzáfér olyan információkhoz, amelyre nincs szüksége
- ha az email, vagy a telefonszám formátuma módosul, az kihatással lesz mindkét kliensre
- ha bármely kliens hatáskörét kiterjesztenénk további elemekre, akkor annak meg kell valósítania a névjegy minden tulajdonságát
- ezért célszerű kiemelni a két funkcionalitást külön interfészekbe (**Emailable**, **Callable**), így az egyes a kliensek csak a számukra szükséges funkcionalitást látják, a névjegy pedig megvalósítja az interfészeket

# Objektumorientált tervezési szempontok és minták

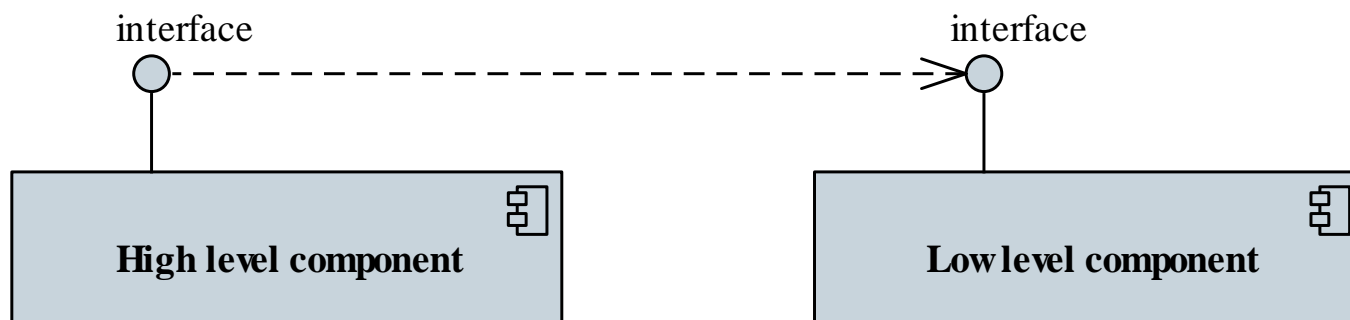
## Interface segregation principle



# Objektumorientált tervezési szempontok és minták

## Dependency inversion principle

- A *Dependency inversion principle* (DIP) a függőségek kezelésével szemben fogalmaz meg elvárás, amelynek értelmében:
  - magasabb szintű programegységek nem függhetnek alacsonyabb szintű programegységtől, hanem mindkettőnek az absztrakciótól kell függenie
  - az absztrakció nem függhet a részletektől, a részletek függenek az absztrakciótól



# Objektumorientált tervezési szempontok és minták

## Dependency inversion principle

- A megvalósítása számára a következőket jelenti:
  - az osztály mezői a konkrét osztályok helyett az absztrakció példányát tartalmazzák
  - a konkrét osztályok az absztrakció segítségével lépnek kapcsolatba egymással, a konkrét osztályok szükség esetén átalakíthatóak
  - szigorú esetben konkrét osztályokból nem örökölhetünk, és már megvalósított metódust nem definiálunk felül
  - az osztályok példányosítását egy külső programegység végzi, pl. függőség befecskendezés, gyártó művelet, vagy absztrakt gyártó segítségével

# Objektumorientált tervezési szempontok és minták

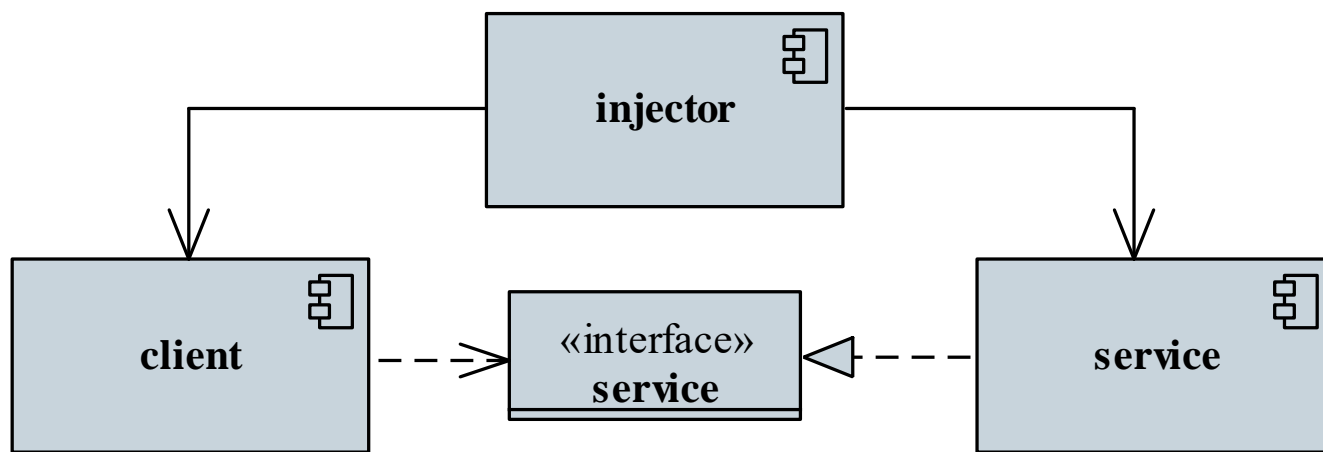
## Függőség befecskendezés

- A végrehajtás során egy réteg (*kliens*) által használt réteg (*szolgáltatás*) egy, az adott körülmények függvényében alkalmazható megvalósítása kerül alkalmazásra
  - erről általában nem dönthet sem a kliens, sem a szolgáltatás, mivel ők nem ismerik a körülményeket, ezért egy külső komponensnek kell megadnia a megvalósítást
- A kliens (*client*) számára a megfelelő szolgáltatás (*service*) külső átadásának egy lehetséges módja a *függőség befecskendezés* (*dependency injection*, DI)
  - a kliens biztosít egy átadási pontot a szolgáltatás interfésze számára, ahol a végrehajtás során a szolgáltatás megvalósítását adhatjuk át

# Objektumorientált tervezési szempontok és minták

## Függőség befecskendezés

- az átadási pont függvényében 3 típusa lehet: *konstruktor*, *metódus* (beállító művelet), *interfész* (a kliens megvalósítja a beállító műveletet)
- a befecskendést végző komponens (*injector*) lehet egy felsőbb réteg, vagy a rétegződéstől független környezeti elem



# Objektumorientált tervezési szempontok és minták

## Függőség befecskendezés

- Pl. :

```
interface Calculator // szolgáltatás interfésze
{
    double compute(double value);
}
...
class LogCalculator implements ICalculator
    // a szolgáltatás egy megvalósítása
{
    public double compute(double value) {
        return Math.Log(value);
    }
}
```



# Objektumorientált tervezési szempontok és minták

## Függőség befecskendezés

- Pl.:

```
class Client { // kliens
    private Calculator calculator; // függőség

    public Client(Calculator s) {
        this.service = s;
    } // konstruktor befecskendezés

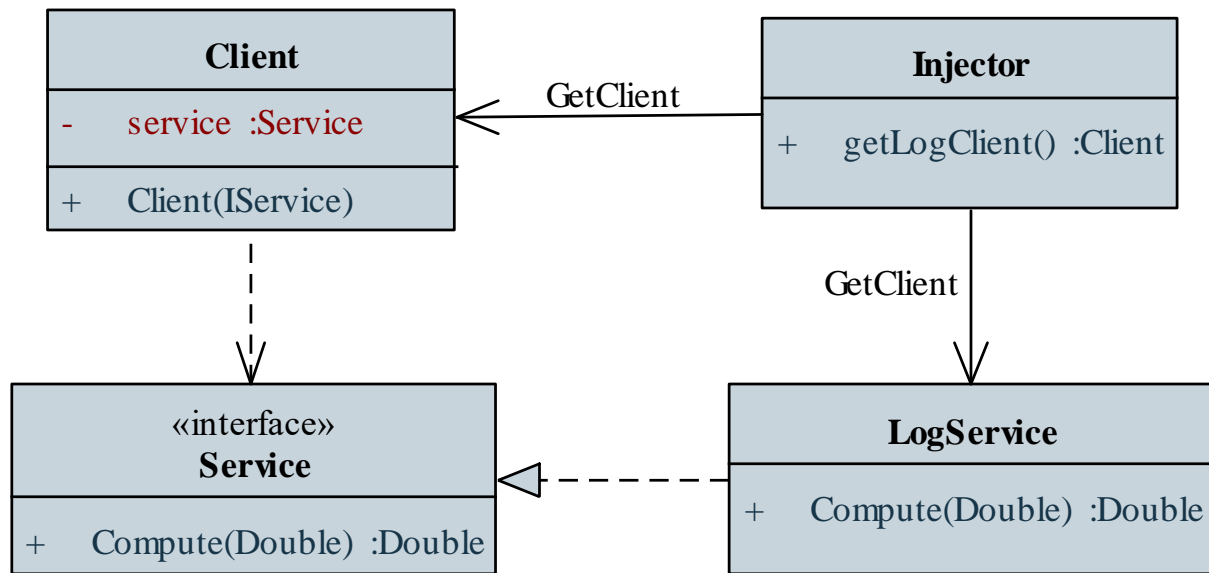
    public void Execute() {
        ...
        v = calculator.Compute(v); // felhasználás
        ...
    }
}
```

# Objektumorientált tervezési szempontok és minták

## Függőségek kezelése

- Pl. :

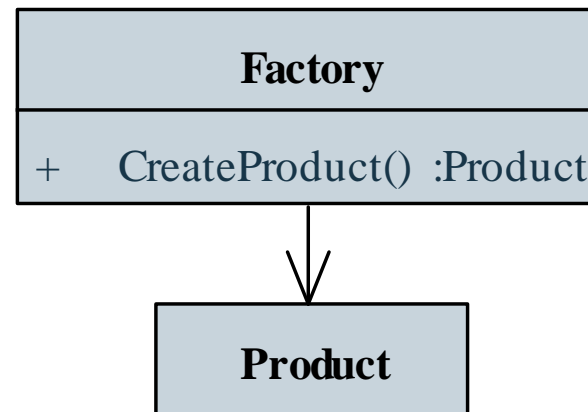
```
class Injector { // befecskendező
    public Client getLogClient() {
        return new Client(new LogService());
    }
}
```



# Objektumorientált tervezési szempontok és minták

## Gyártó

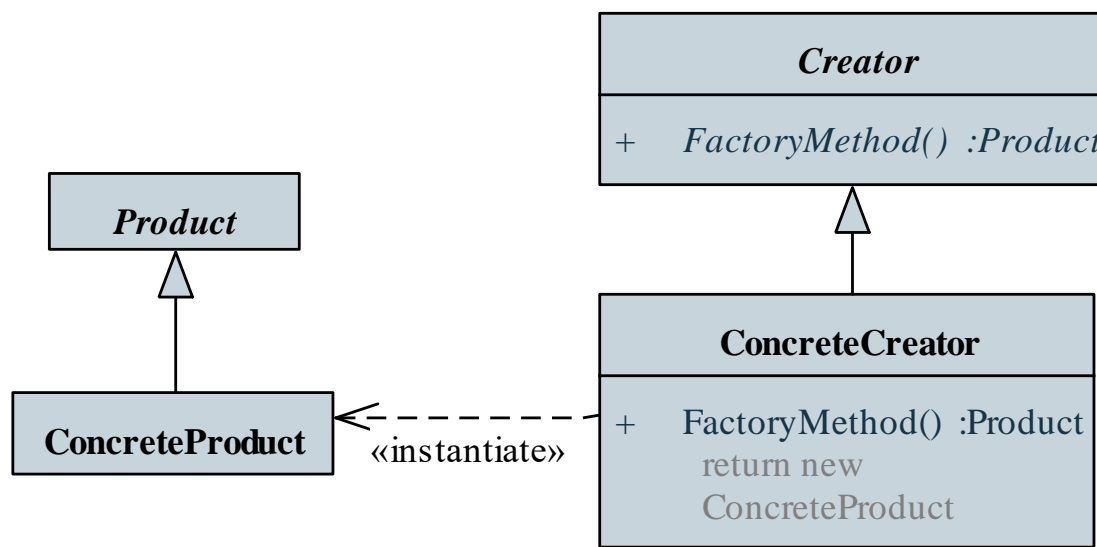
- A *gyártó (factory)* egy olyan objektum, amelynek feladata más objektumok előállítása műveletek segítségével
  - lehetőséget ad objektumok példányosításánál:
    - a konkrét példányosítandó típus kiválasztására
    - a példányosítással járó kódismétlés kiküszöbölésére
    - fel nem fedhető környezeti információk használatára
    - a példányosítási lehetőségek bővítésére
  - lehetővé teszi a konkrét leggyártott típus cseréjét a gyártó művelet és az absztrakt gyártó segítségével



# Objektumorientált tervezési szempontok és minták

## Gyártó művelet

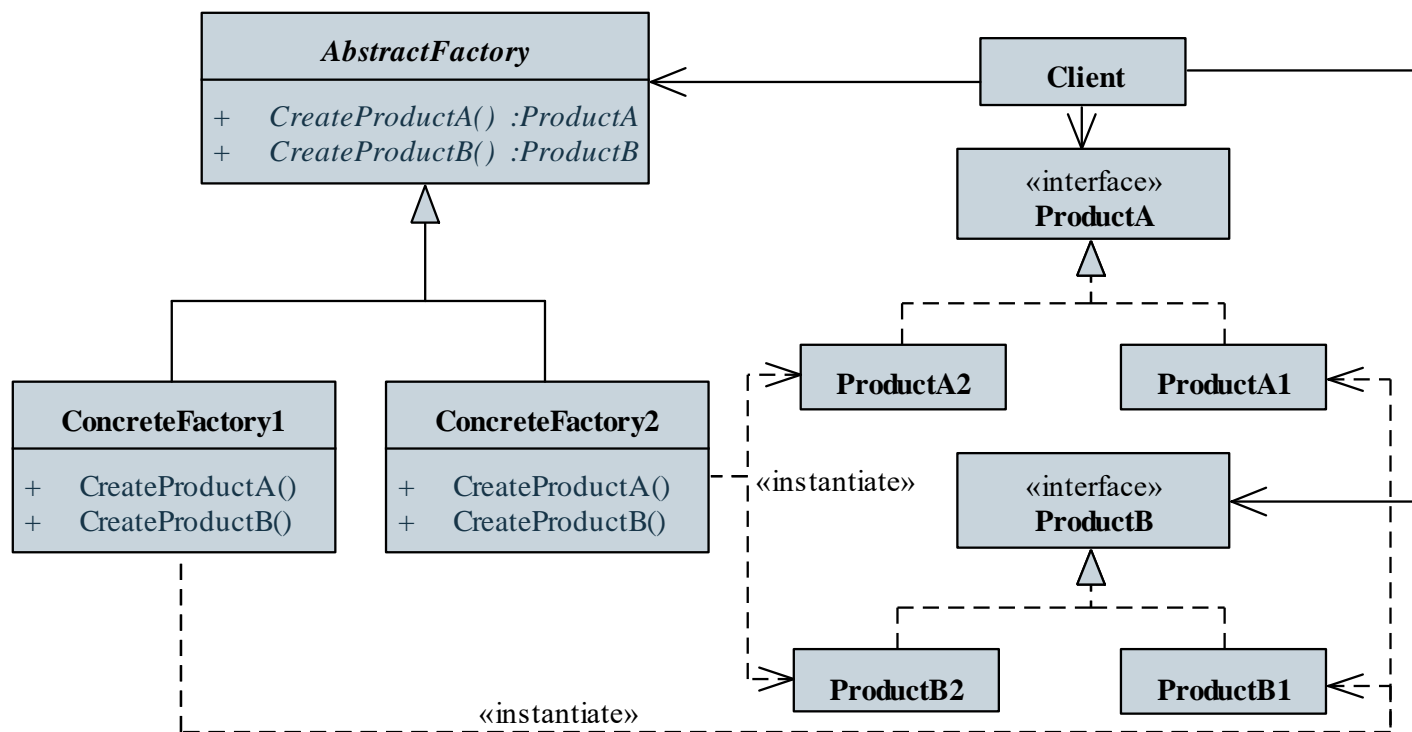
- Amennyiben egy típus konkrét megvalósítását szeretnénk befolyásolni, a *gyártó művelet* (*factory method*) tervmintát használhatjuk
  - a gyártó művelet (**FactoryMethod**) felüldefiniálásával bármely leszármazott terméket előállíthatjuk



# Objektumorientált tervezési szempontok és minták

## Absztrakt gyártó

- Amennyiben több, kapcsolatban álló típus konkrét megvalósítását szabályoznánk, az *absztrakt gyártó* (*abstract factory*) tervmintát használhatjuk



# Objektumorientált tervezési szempontok és minták

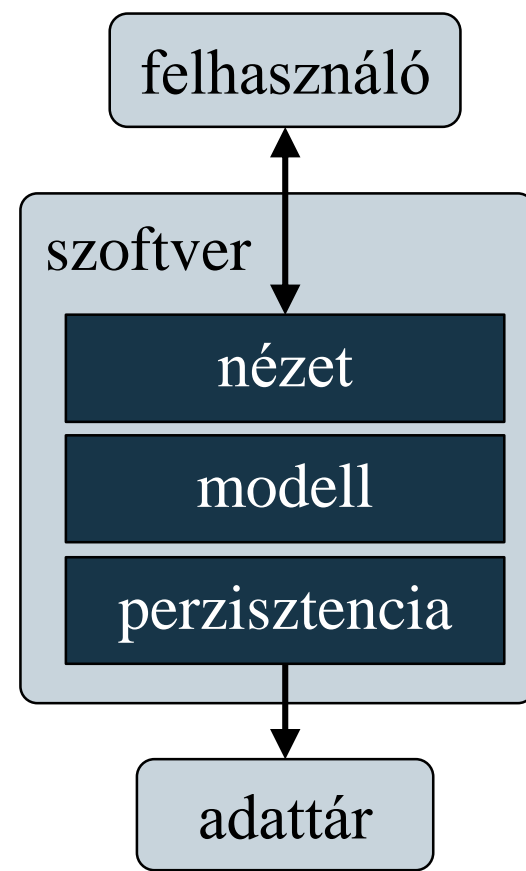
## Az alapelvek hatása az architektúrára

- Az alapelveket a tervezés minden szintjén, így az architektúra szintjén is célszerű betartani
  - *Monolitikus architektúra* esetén egyáltalán nem érvényesülnek az alapelvek
  - *Modell-nézet architektúra* esetén is túl nagy a komponensek felelőssége:
    - a *modell* felel az üzleti logikáért (programállapot kezelése, algoritmusok végrehajtása), valamint az adatok hosszú távú tárolásáért (mentés, betöltés)
    - a *nézet* felel az adatok megjelenítéséért, valamint a vezérlésért (felhasználó tevékenység fogadása és feldolgozása)

# Objektumorientált tervezési szempontok és minták

## Háromrétegű architektúra

- A modell felbontásával jutunk a *háromrétegű* (3-tier) architektúrához, amelyben elkülönül:
  - a *nézet* (*presentation, view, display*)
  - a *modell* (*logic, business logic, application, domain*), amely tartalmazza az állapotkezelést, valamint az algoritmusok
  - a *perzisztencia*, vagy *adatelérés* (*data, data access, persistence*), amely biztosítja a hosszú távú adattárolás (pl. fájl, adatbázis) kezelését



# Objektumorientált tervezési szempontok és minták

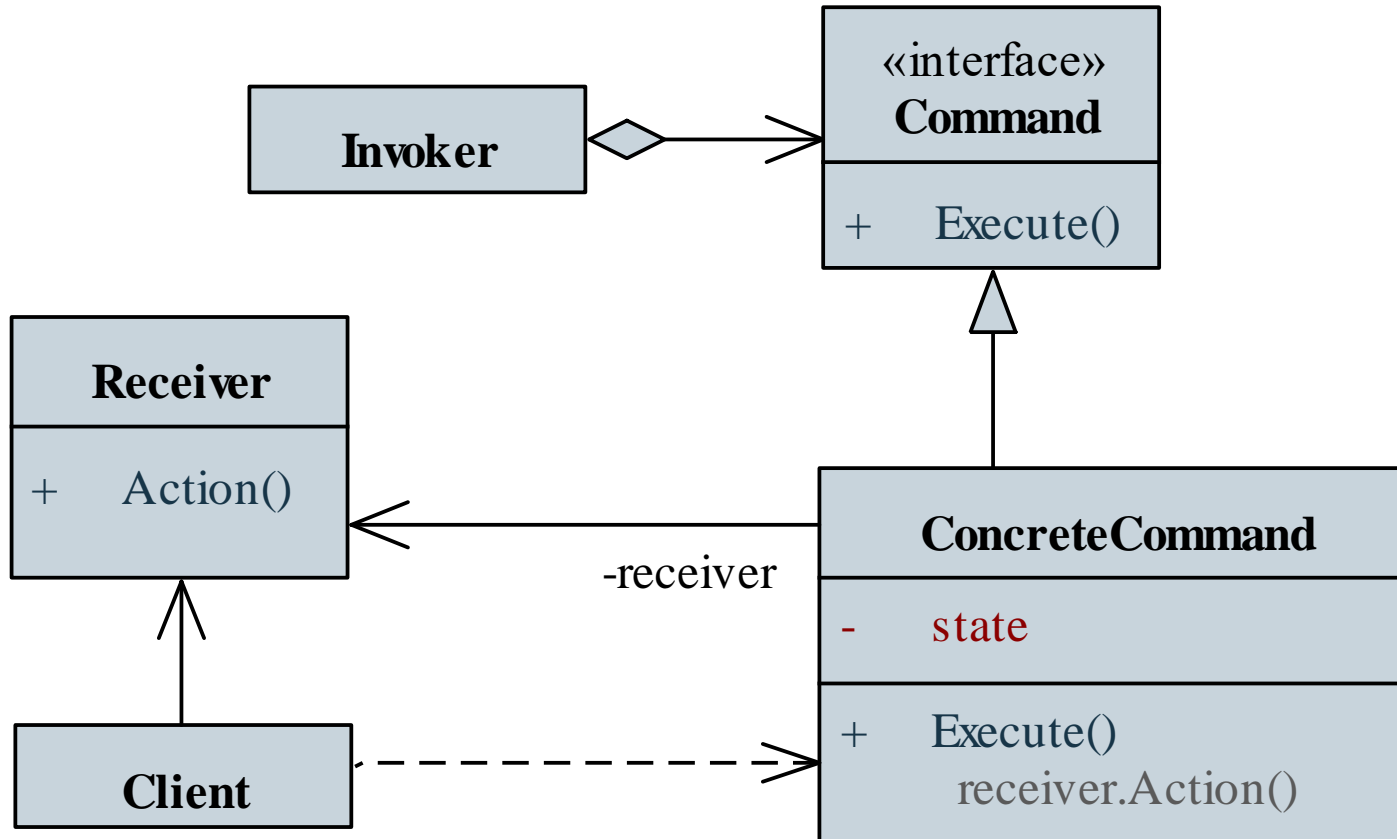
## Parancs tervminta

- A háromrétegű architektúrában a nézet továbbra is összetett, de a tevékenységek leválaszthatóak, a *parancs (command)* tervminta lehetőséget ad egy művelet kiemelése egy külön objektumba
  - a végrehajtandó tevékenység (**Action**) formája, paraméterezése tetszőleges lehet, ezért nem lehet egyazon módon különböző környezetekben kezelni
  - a parancs (**Command**) egy egységes formát biztosít egy tevékenység végrehajtására (**Execute**), a konkrét parancs (**ConcreteCommand**) pedig végrehajtja a tevékenységet
  - a kezdeményező (**Invoker**) csupán a parancsot látja, így a tényleges végrehajtás előle rejtett marad



# Objektumorientált tervezési szempontok és minták

## Parancs tervminta



# Objektumorientált tervezési szempontok és minták

## Adatok állapotai

- A háromrétegű alkalmazásokban az adatok három állapotban jelennek meg
  - *megjelenített állapot (display state)*: a felhasználói felületen megjelenő tartalomként, amelyet a felhasználó módosíthat
  - *munkafolyamat állapot (session state)*: a memóriában, amely mindaddig elérhető, amíg a program és felhasználója aktív
  - *rekord állapot (record state)*: az adat hosszú távon megőrzött állapota az adattárban (perzisztenciában)



# Objektumorientált tervezési szempontok és minták

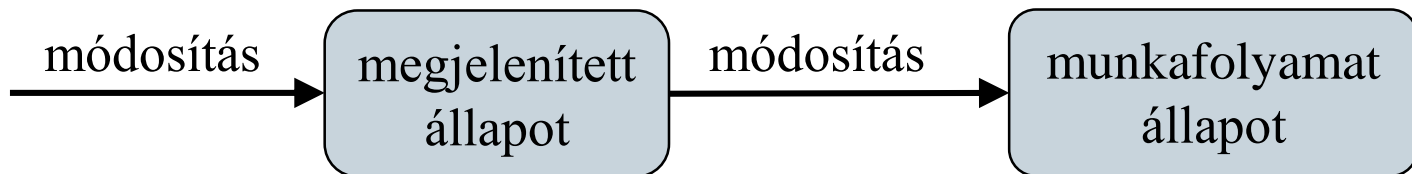
## Adatkötés

- Az állapotok között a programnak *transzformációkat* és *szinkronizációt* kell végeznie
  - a nézet állapot és a munkafolyamat állapot sokszor megegyezik, a munkafolyamat és a perzisztens állapot (adattárolási forma) sokszor különbözik, ezért külön adatelérést kell biztosítanunk
  - a munkafolyamat és a perzisztens állapot között általában ritkán történik szinkronizáció (pl. program indítása, leállítása)
  - a nézet állapot és a munkafolyamat állapotot rendszerint állandó szinkronban kell tartanunk, ezt lehetőség szerint automatizálnunk kell

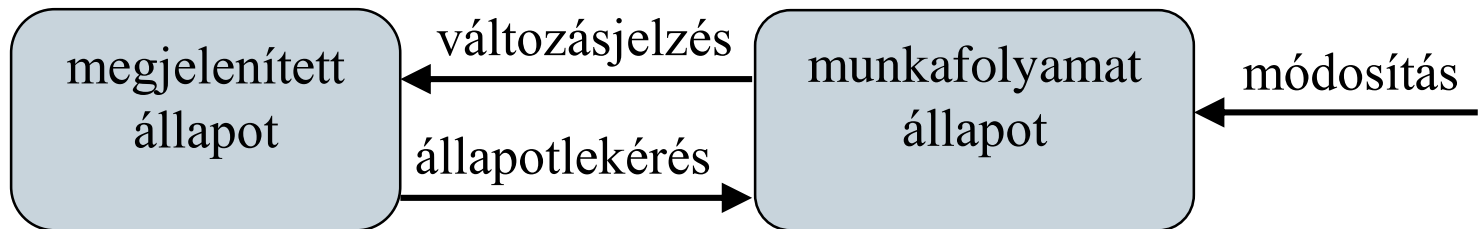
# Objektumorientált tervezési szempontok és minták

## Adatkötés

- Az állapotok automatikus szinkronizálását *adatkötés* (*data binding*) segítségével érhetjük el, amely két lehetséges módon biztosíthatja a szinkronizációt
  - az érték módosítás hatására átíródhat a másik állapotba



- az érték módosítás hatására jelzést adhat a másik állapot frissítésére



# Objektumorientált tervezési szempontok és minták

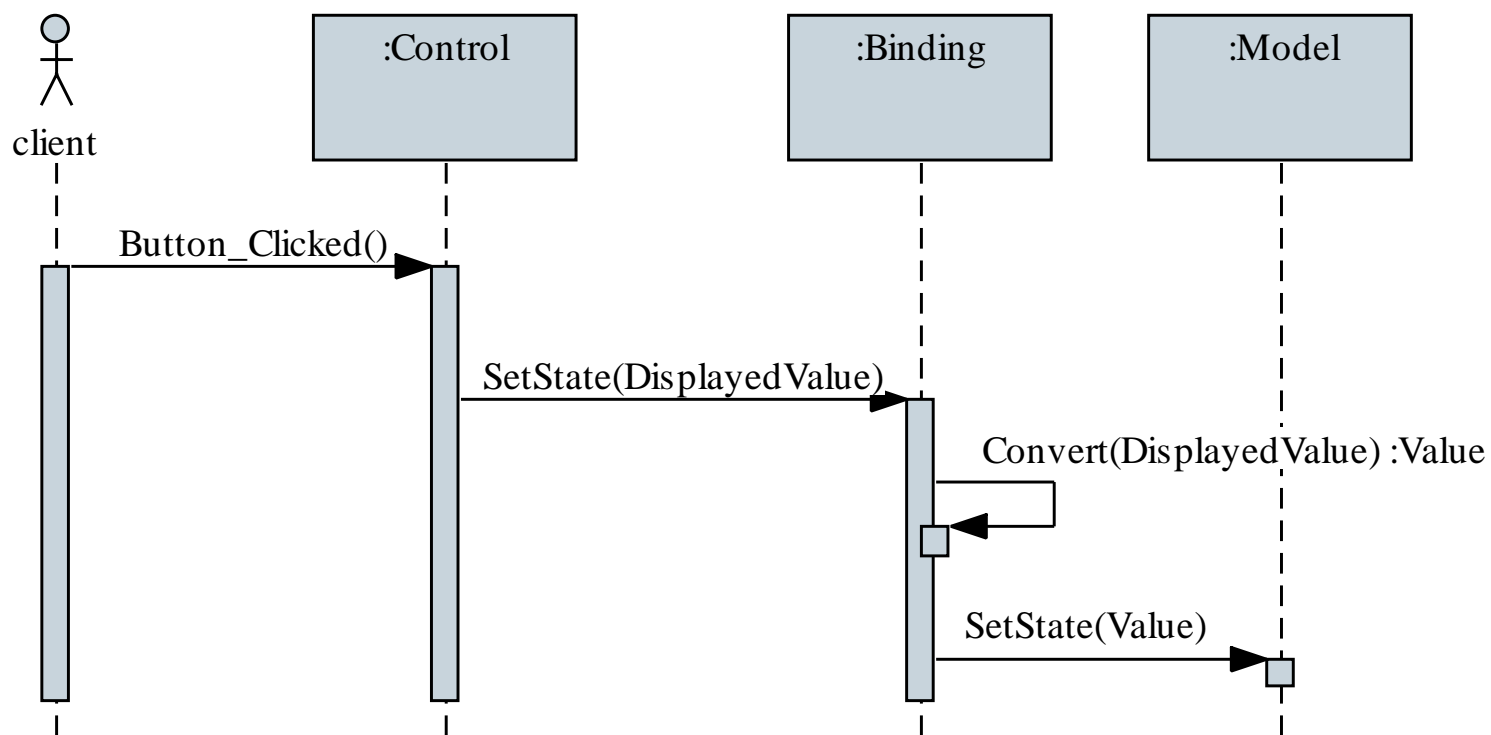
## Adatkötés

- Az adatkötésért egy olyan adatkötés objektum (**Binding**) felel, amelyet a megjelenített állapot tárolója (nézet) és a munkafolyamat állapot tárolója (modell) közé helyezünk
  - az adatkötés ismeri mind a nézetet mind a modellt, ezért lehívhatja az értékeket (**GetState**), és elvégezheti a módosítást (**SetState**)
  - elvégezheti az átalakítást (**Convert**) a munkafolyamat állapot és a nézet állapot között
  - általában minden szinkronizálandó adathoz külön adatkötés tartozik, többszörös előfordulás esetén akár előfordulásonként is tartozhat adatkötés

# Objektumorientált tervezési szempontok és minták

## Adatkötés

- a nézet ismerheti az adatkötést, így közvetlenül kezdeményezheti a módosítást az adatkötésen keresztül



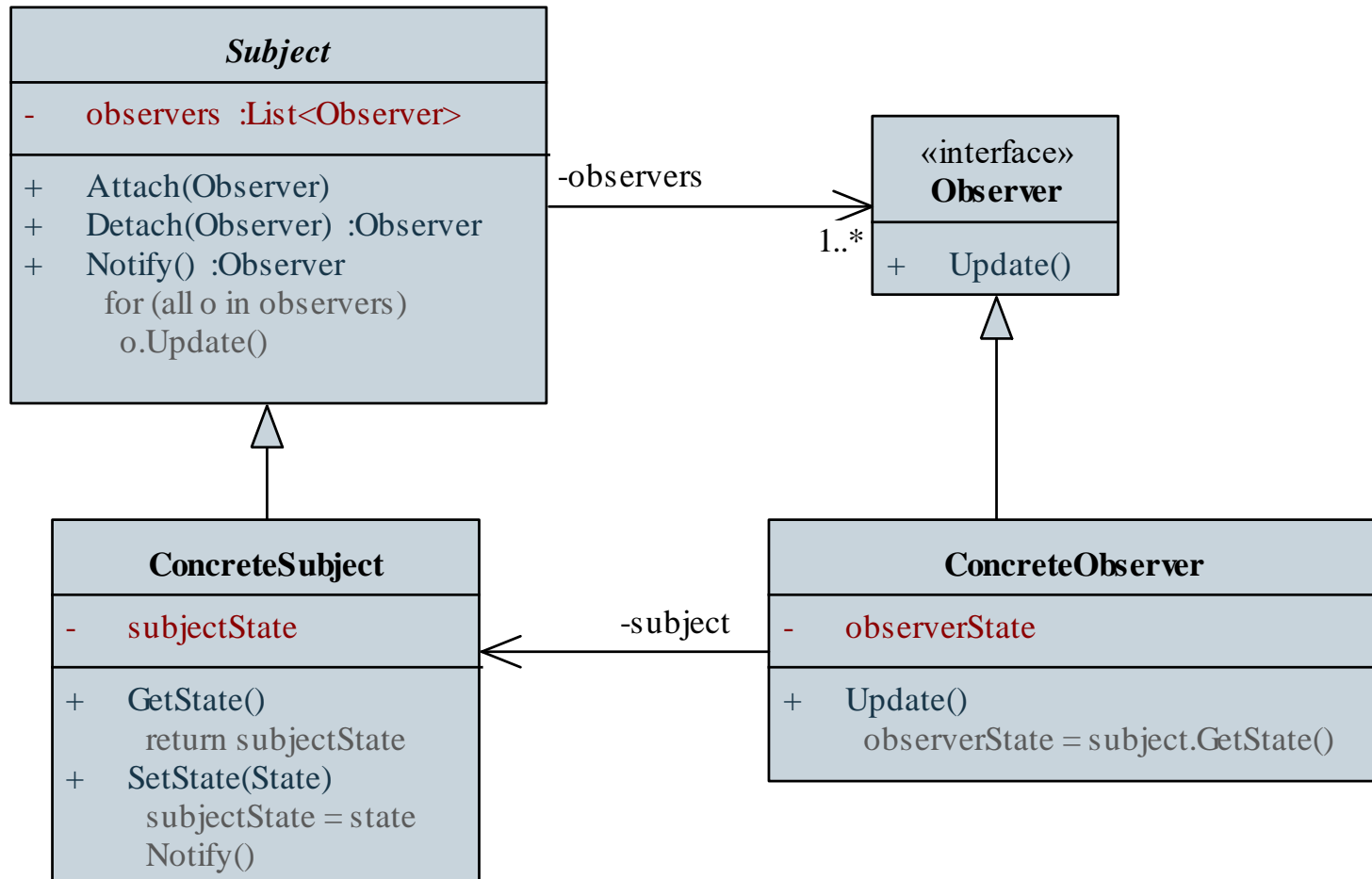
# Objektumorientált tervezési szempontok és minták

## Adatkötés

- a modell azonban nem ismerheti az adatkötést, sem a vezérlőt, csak jelzést tud adni az állapot változásáról, ennek feldolgozását *figyelő tervminta* segítségével végezzük
- A *figyelő (observer)* tervminta célja összefüggőség megadása az objektumok között, hogy egyik állapotváltozása esetén a többiek értesítve lesznek
  - a figyelő (**Observer**) ehhez biztosítja a változás jelzésének metódusát (**Update**)
  - a megfigyelt objektumok (**Subject**) az értékeikben tett változtatásokat jelzik a felügyelőknek (**Notify**)
  - egy objektumot több figyelő is nyomon kísérhet

# Objektumorientált tervezési szempontok és minták

## Figyelő tervminta

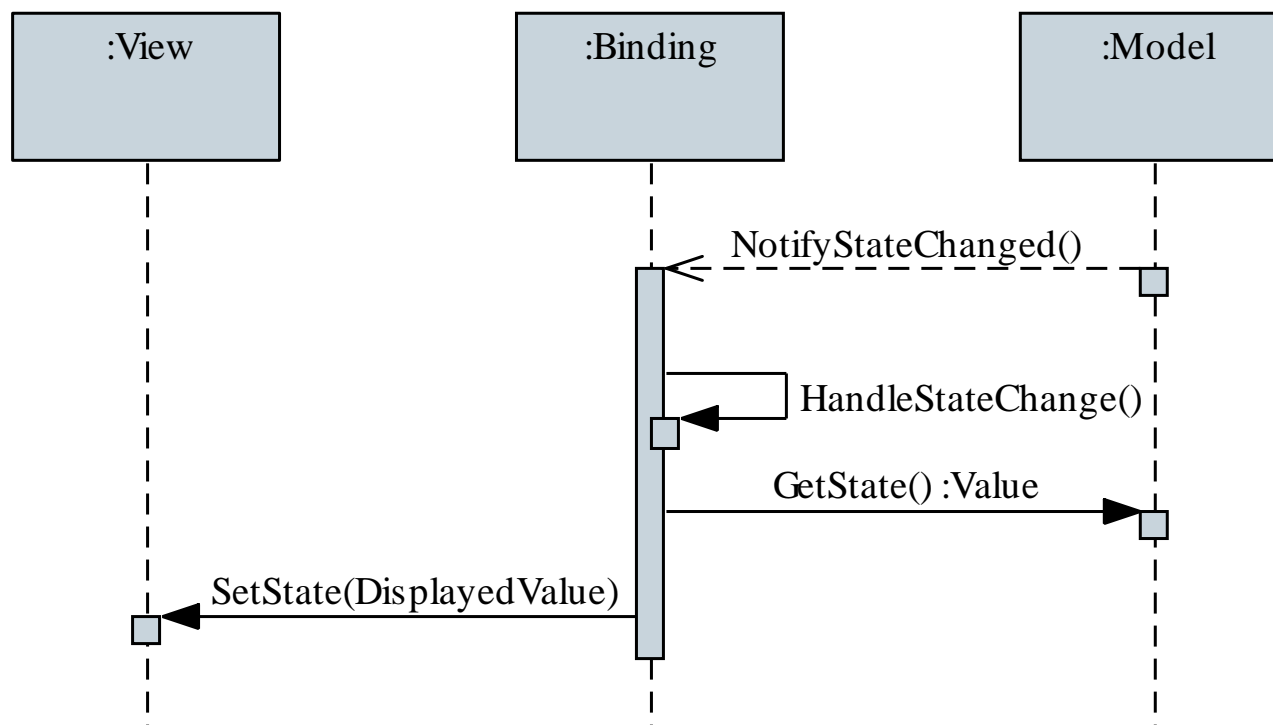




# Objektumorientált tervezési szempontok és minták

## Adatkötés megvalósítása

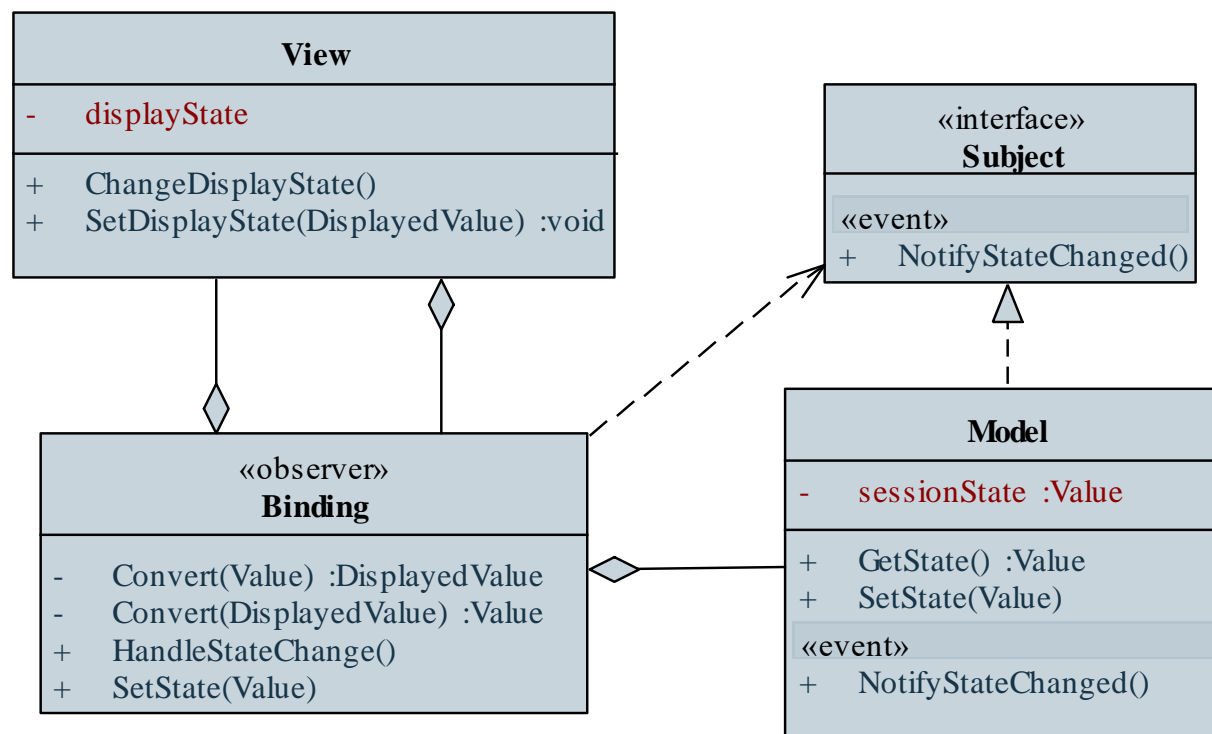
- Az adatkötés megvalósítja a figyelőt, ám mivel közvetlenül nem ismerheti a modellt, a változás jelzése esemény formájában történik (`NotifyStateChanged`)



# Objektumorientált tervezési szempontok és minták

## Adatkötés megvalósítása

- mivel a jelzésnek egységes formában kell megvalósulnia, célszerű, ha minden modell egy közös, az esemény kiváltását biztosító interfészt valósít meg (**Subject**)



# Objektumorientált tervezési szempontok és minták

## Adatkötés

- Előnyei:
  - a megjelenített és a munkafolyamat állapot mindig szinkronban tartható, automatikusan
- Hátrányai:
  - a szinkronizáció erőforrásigényes, sokszor feleslegesen hajtódik végre
  - a rendszernek ügyelnie kell a körkörös módosítások elkerülésére
  - a modellnek biztosítania kell a megfelelő interfész megvalósítását, és az esemény kiváltását a szükséges pontokon

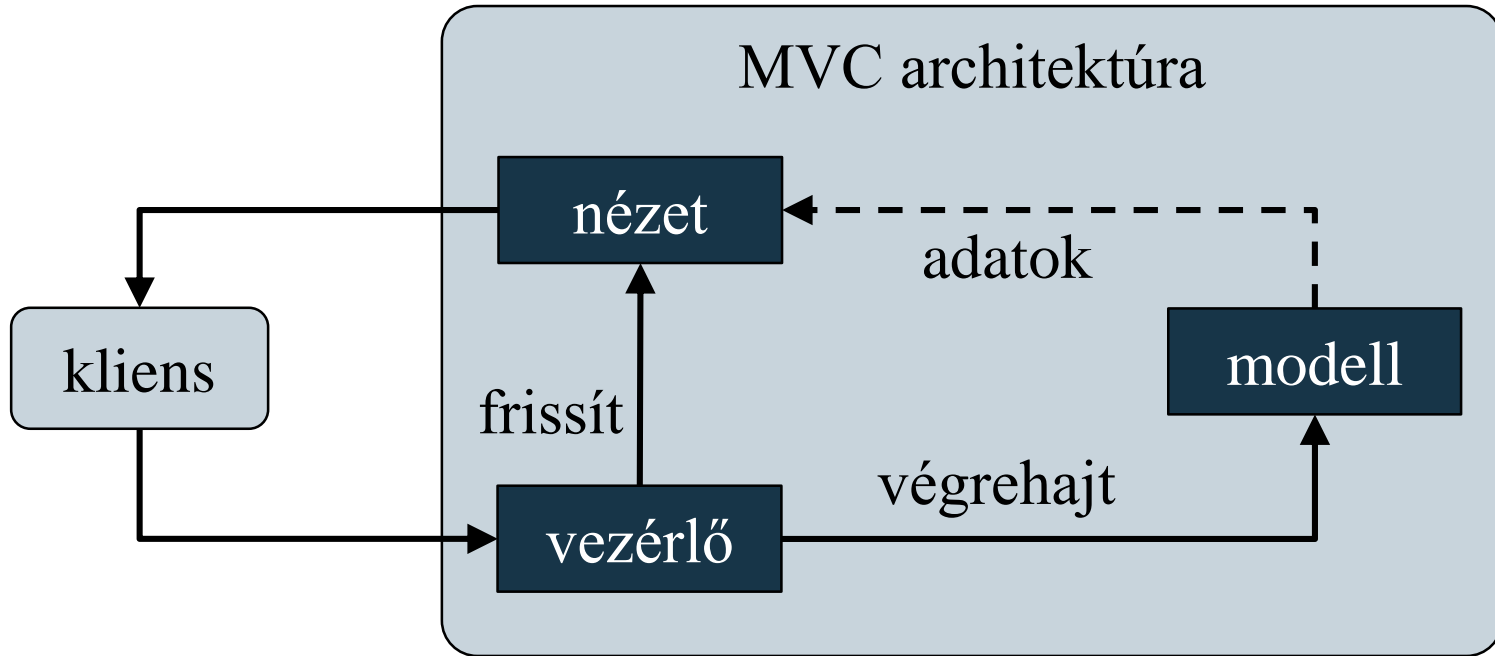
# Objektumorientált tervezési szempontok és minták

## MVC architektúra

- A megjelenítés és a vezérlés leválasztását biztosítja a *Model-View-Controller (MVC)* architektúra, amelyben
  - a *vezérlő (Controller)* fogadja közvetlenül a kérést a felhasználótól, feldolgozza azt (a modell segítségével), majd előállítja a megfelelő (új) nézetet
  - a *nézet (View)* a felület (jórészt deklaratív) meghatározása, amely megjeleníti, illetve átalakítja az adatokat
    - *lehívás alapú (pull-based)*: ismeri a modellt, az adatokat a modelltől kéri le
    - *betöltés alapú (push-based)*: a vezérlő adja át számára az adatokat
  - a modell mellett perzisztációval is bővíthető a szerkezet

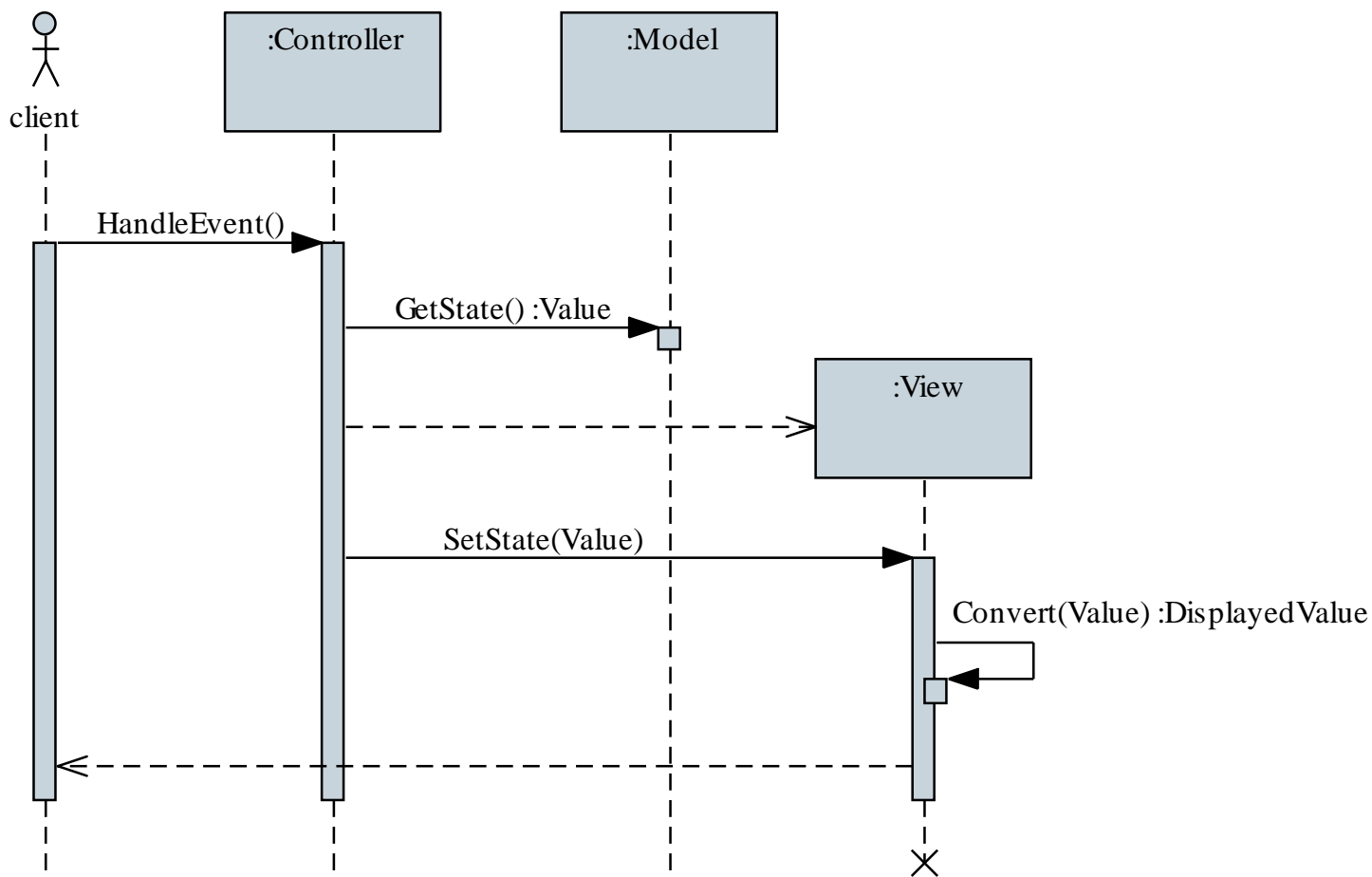
# Objektumorientált tervezési szempontok és minták

## MVC architektúra



# Objektumorientált tervezési szempontok és minták

## MVC architektúra (betöltés alapú)



# Objektumorientált tervezési szempontok és minták

## MVC architektúra

- Az MVC architektúra különösen webes környezetben népszerű, ahol könnyen leválasztható a nézet (HTML) a vezérléstől (URL)
  - általában egy vezérlőhöz több nézet is tartozik
- Előnyei:
  - a nézetnek nem kell foglalkoznia az események feldolgozásával, mivel azokat közvetlenül a vezérlő kapja meg
- Hátrányai:
  - a nézet továbbra is felel az adatok megfelelő átalakításáért, tehát tartalmaz valamennyi logikát