

# Unified Modeling Language

---

Szoftvertechnológia

Dr. Simon Balázs

BME, IIT

# Tartalom

---

- UML diagrammok:
  - osztálydiagram
  - csomagdiagram
  - objektumdiagram

# Hol tartunk?



Use Case diagram

← A funkcionális követelmények magas szintű leírása:  
A use case-ek alapvető interakciós sorozatok a rendszer és a felhasználói között

Aktivitásdiagram

← Use case interakciós sorozatok munkafolyamatként ábrázolva

Komponens-  
diagram

← Áttekintő kép a rendszer architektúrájáról:  
A komponensek és kapcsolataik

Hova kell telepíteni a komponenseket

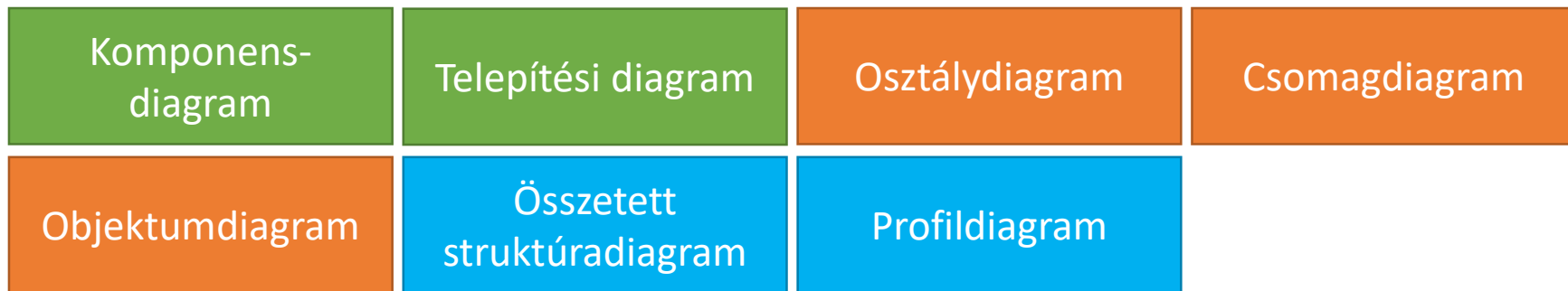
→ Telepítési diagram

Most következik:  
How to design the inside of a component?  
(Struktúra és viselkedés)

# Hol tartunk?

Most következik:  
Hogyan tervezzük meg egy komponens belsejét?  
(Struktúra és viselkedés)

## Strukturális UML diagramok:



## Viselkedési UML diagramok:



# Osztálydiagram (Class Diagram)

---

# Osztálydiagram

---

- Objektumorientált modellek leírásának szabványos módja
- A leggyakrabban használt UML diagram
- Tipikusan szoftverfejlesztők használják a szoftver architektúrájának dokumentálására
- Az osztálydiagramok közvetlenül programkóddá alakíthatók
- Vigyázat:
  - Az osztálydiagram független a programozási nyelvektől
  - Néhány fogalomnak más a jelentése UML-ben, mint egy konkrét programozási nyelven

# Osztálydiagram

- Az osztálydiagram elemei:
  - **Osztály (Class)**
    - objektumok közös viselkedését, kényszereit és szemantikáját írja le
    - viselkedés: **Operációk (operations)** (aka. metódusok (methods))
    - állapot: **Tulajdonságok (properties)** (aka. mezők (fields) vagy attribútumok (attributes))
  - **Interfész (Interface)**
    - publikus **Operációk** halmaza, amelyek egy adott viselkedést és kényszereket írnak elő
    - nem definiál implementációt
    - az interfész által előírt viselkedést egy osztály implementálhatja
  - Osztályok és interfészek közös neve: **classifier**
- Kapcsolatok az elemek között:
  - **Megvalósítás (realization)**: egy osztály meg tud valósítani (implementálni tud) egy interfészt
  - **Öröklődés (generalization)**: egy osztály/interfész leszármazhat egy másik osztályból/interfészből
  - **Asszociáció (association)**: egy classifier hivatkozhat egy másik classifier-re
  - **Függőség (dependency)**: egy classifier függhet egy másik classifier-től

# Osztály

- Közös viselkedéssel bíró objektumokat ír le
- Az osztály példányainak viselkedése:  
**Operációk (operations)** (aka. metódusok (methods))
- Az osztály példányainak állapota:  
**Tulajdonságok (properties)** (aka. mezők (fields) vagy attribútumok (attributes))

Név compartment	{	<b>PacMan</b>
Attribútum compartment	{	lives: int points: int
Operáció compartment	{	Move(d:Direction):void Die():void

C++/Java/C# leképezés:

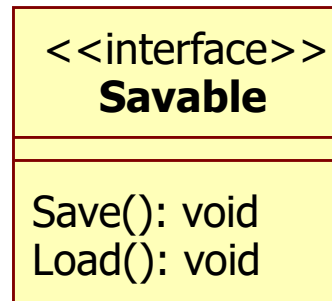
```
class PacMan {  
    int lives;  
    int points;  
  
    void Move(Direction d) {  
        // ...  
    }  
    void Die() {  
        // ...  
    }  
}
```



# Interfész

- Publikus **Operációk** halmaza, amelyek egy adott viselkedést és kényszereket írnak elő

Jelölés: olyan, mint egy osztály <<interface>> sztereotípiával:



C++ leképezés:

```
class Savable {
public:
    void Save() = 0;
    void Load() = 0;
}
```

Java/C# leképezés:

```
interface Savable {
    void Save();
    void Load();
}
```

# Láthatóságok

- Az operációknak és tulajdonságoknak vannak láthatóságaik
- Láthatóságok:
  - **private** (-): csak az adott osztály tagjai számára látható
  - **protected** (#): csak az adott osztály és a leszármazottak tagjai számára elérhető
  - **public** (+): bárki számára elérhető, aki az osztályt eléri
  - **package** (~): bárki számára elérhető, aki ugyanabban a csomagban van, mint az osztály
- **Figyelem: az UML által definiált láthatóságok jelentése eltérhet a programnyelvekben definiált láthatóságoktól**
  - C++: nincs *package* láthatóság, de van *friend*
  - Java: a *protected* egyben *package* is
  - C#: nincs *package* láthatóság, de van *internal*

# Láthatóság példa

Foo
-Bar(): void #Baz(): void +Quux(): void ~Garply(): void

Java leképezés:

```
public class Foo {  
    private void Bar() { /* ... */ }  
    protected void Baz() { /* ... */ }  
    public void Quux() { /* ... */ }  
    void Garply() { /* ... */ }  
}
```

C++ leképezés:

```
class Foo {  
private:  
    void Bar() { /* ... */ }  
    void Garply() { /* ... */ }  
    friend ...  
protected:  
    void Baz() { /* ... */ }  
public:  
    void Quux() { /* ... */ }  
};
```

C# leképezés:

```
public class Foo {  
    private void Bar() { /* ... */ }  
    protected void Baz() { /* ... */ }  
    public void Quux() { /* ... */ }  
    internal void Garply() { /* ... */ }  
}
```

# Operációk (operations)

- Az osztály példányainak viselkedését írják le: az objektumok által biztosított szolgáltatásokat
- Egy operáció szignatúrája:

Láthatóság      Paraméterek vesszővel elválasztva      Visszatérési típus

↓      ↙      ↘

+Add(left: double, right: double): double

↑      ↗      ↖

Operáció neve      Paraméter neve      Paraméter típusa

- Opcionális elemek: láthatóság, paraméterek típusa, visszatérési típus
  - alapértelmezett értékük: nem definiált
- A grafikus tervezőeszközök elrejthetnek egyes elemeket ezek közül a jobb áttekinthetőség kedvéért, de ez nem feltétlenül azt jelenti, hogy ezek az elemek hiányoznak

# Tulajdonságok (properties)

- Az osztály példányainak állapotát írják le: az objektumok lehetséges állapotait
- Egy tulajdonság szignatúrája:

Láthatóság      Multiplicitás

↓                      ↓

`-name: String[0..*] = null`

↑                      ↑                      ↓

Tulajdonság neve    Típus                      Alapértelmezett érték

- Opcionális elemek: láthatóság, típus, multiplicitás, alapértelmezett érték
  - a láthatóság, típus, alapértelmezett érték alapértelmezett értéke: nem definiált
  - multiplicitás alapértelmezett értéke: 1 (pontosan 1)
- A grafikus tervezőeszközök elrejthetnek egyes elemeket ezek közül a jobb áttekinthetőség kedvéért, de ez nem feltétlenül azt jelenti, hogy ezek az elemek hiányoznak

# Példány szintű tagok

---

- Tagfüggvények (member operation) és -tulajdonságok (member property) az osztály példányaihoz kapcsolódnak
  - példány-szintűnek (instance-scope) is hívják őket
- A tagtulajdonságok értékei példányonként különböznek
  - ha az egyik példányban megváltozik az értékük, az nincs kihatással a többi példányra
- A tagfüggvények tipikus implementációja a programnyelvekben: implicit nulladik paraméter (this/self)
  - a this/self pointer jelöli az objektumot, amin a függvény meghívták
  - a tagfüggvények és -tulajdonságok elérhetőek ezen a this/self pointeren keresztül

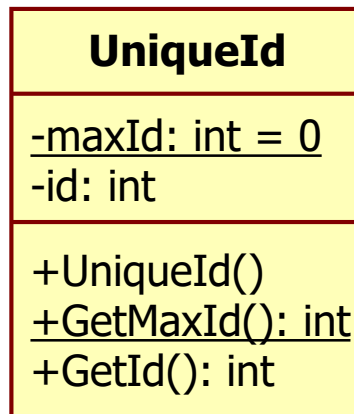
# Statikus tagok

---

- Statikus függvények (static operations) és -tulajdonságok (static properties) az osztályhoz kapcsolódnak
  - osztály-szintűnek (class-scope) is hívják őket
- A statikus tulajdonságok értékei közösek az összes példányra nézve
  - ha egy példányban megváltozik az értékük, az összes többi példány is ugyanezt az értéket fogja látni
- Egy statikus függvénynek nincs this/self pointere
  - közvetlenül nem érhetők el belőle a tagfüggvények és -tulajdonságok
- Statikus függvényeket nem lehet felülírni a leszármazottakban

# Statikus példa

- A statikus tagok jelölése aláhúzással történik:



C++ leképezés:

```
class UniqueId {
private:
    static int maxId;
    int id;
public:
    UniqueId() {
        this->id = ++UniqueId::maxId;
    }
    static int GetMaxId() {
        return UniqueId::maxId;
    }
    int GetId() {
        return this->id;
    }
};

int UniqueId::maxId = 0;
```

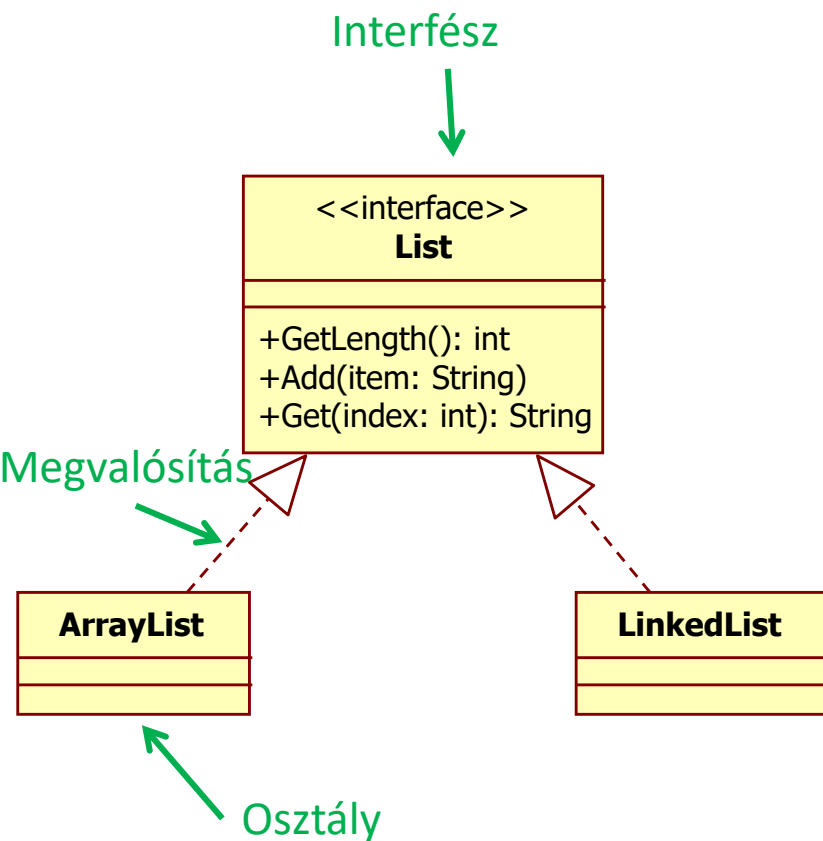
Java/C# leképezés:

```
public class UniqueId {
    private static int maxId = 0;
    private int id;
    public UniqueId() {
        this.id = ++UniqueId.maxId;
    }
    public static int GetMaxId() {
        return UniqueId.maxId;
    }
    public int GetId() {
        return this.id;
    }
}
```



# Megvalósítás (realization)

- Osztályok implementálhatnak/megvalósíthatnak (realize) interfészeket



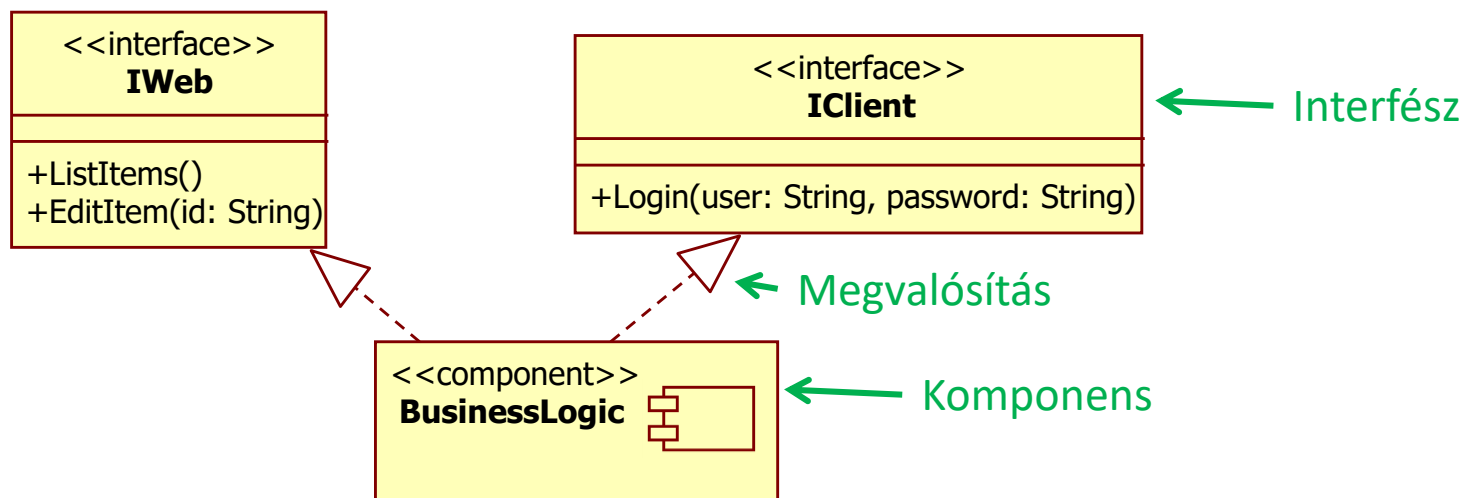
Java leképezés:

**Megvalósítás** (Realization) is indicated by a green arrow pointing to the `implements` keyword in the Java code.

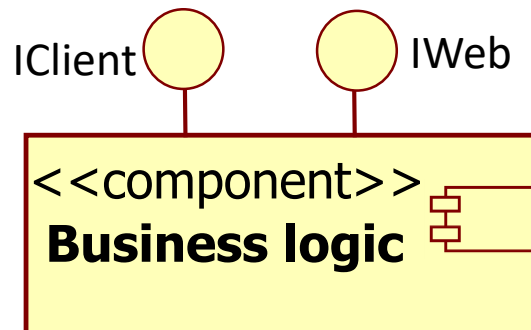
```
public class ArrayList implements List
{
    public int GetLength()
    {
        // ...
    }
    public void Add(String item)
    {
        // ...
    }
    public String Get(int index)
    {
        // ...
    }
}
```

# Megvalósítás (realization)

- Komponensek is megvalósíthatnak (realize) interfészeket
  - ezeket az interfészeket biztosítja az interfész



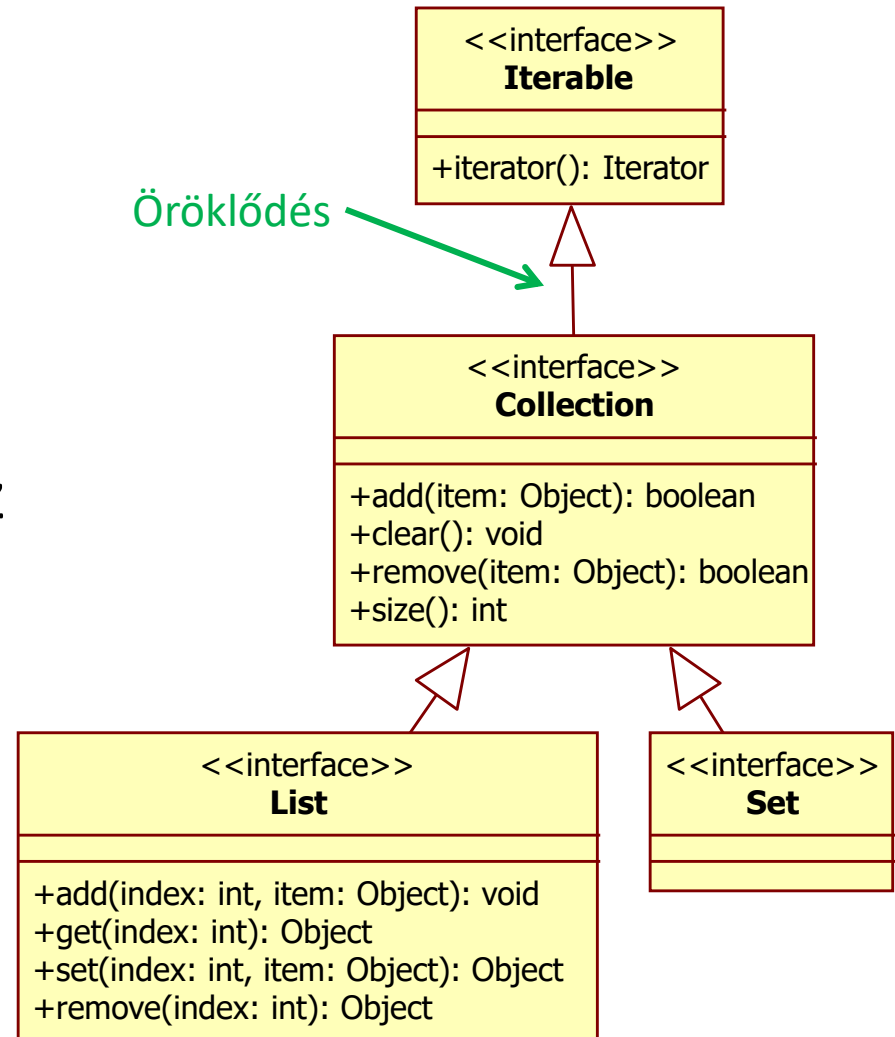
Ekvivalens a nyalóka jelöléssel:  
a nyalóka jelölésnél azonban nem  
mutatjuk az interfész operációit



# Öröklődés (generalization)

- Öröklődés interfészek között
  - többszörös is megengedett
- Az öröklődés “az-egy” reláció a leszármazott és az ős interfész között
- A leszármazott interfész újrahasznosítja és kibővíti az ős által definiált viselkedést

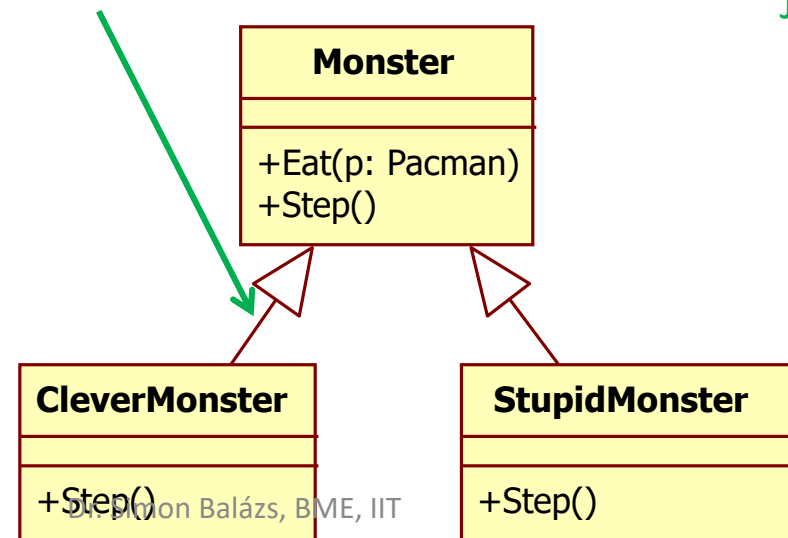
Példa: Java kollekciók



# Öröklődés (generalization)

- Öröklődés osztályok között
  - többszörös is megengedett
- Az öröklődés “az-egy” reláció a leszármazott és az ős osztály között
- A leszármazott osztály újrahasznosítja és kibővíti az ős által definiált viselkedést
- Virtuális metódus (virtual method):
  - virtuális metódusokat felüldefiniálhatnak a leszármazottak
  - így lehet kiterjeszteni az ős viselkedését
  - az UML-ben minden tagfüggvény virtuális

Öröklődés

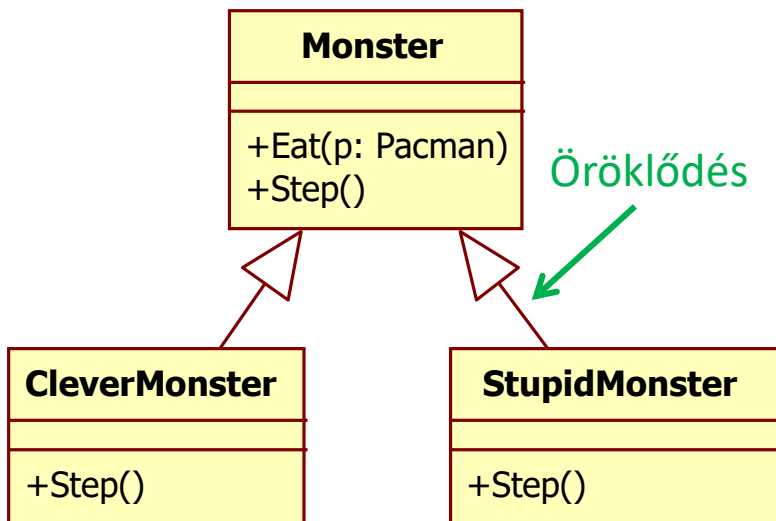


Java leképezés:

```
public class Monster
{
    public void Eat(Pacman p) { /*...*/ }
    public void Step() { /*...*/ }
}

public class CleverMonster extends Monster
{
    public void Step() { /*...*/ }
}
```

# Öröklődés (generalization)



C++ leképezés:

```
class Monster
{
    public void Eat(Pacman* p) { /*...*/ }
    public virtual void Step() { /*...*/ }
}
class CleverMonster : public Monster
{
    public void Step() { /*...*/ }
}
```

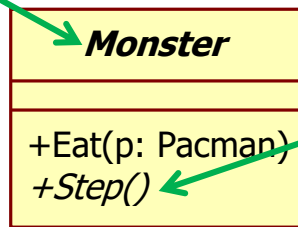
C# leképezés:

```
public class Monster
{
    public void Eat(Pacman p) { /*...*/ }
    public virtual void Step() { /*...*/ }
}
public class CleverMonster : Monster
{
    public override void Step() { /*...*/ }
}
```

# Absztrakt operációk és absztrakt osztályok

- Absztrakt operáció (abstract operation):
  - virtuális függvény implementáció nélkül
  - egy konkrét (nem absztrakt) leszármazottnak kell hogy legyen implementációja erre a függvényre
- Absztrakt osztály (abstract class):
  - nem példányosítható
  - általában van legalább egy absztrakt függvénye, de ez nem követelmény
- Jelölés: dőlt betű

Absztrakt  
osztály



Absztrakt  
metódos

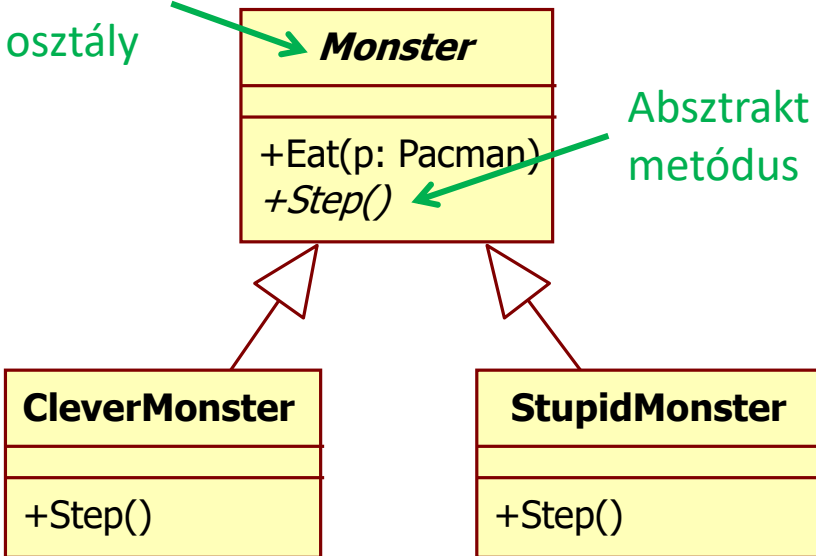
Java leképezés:

```
public abstract class Monster
{
    public void Eat(Pacman p) { /*...*/ }
    public abstract void Step();
}

public class CleverMonster extends Monster
{
    public void Step() { /*...*/ }
}
```

# Absztrakt operációk és absztrakt osztályok

Absztrakt osztály



C++ leképezés:

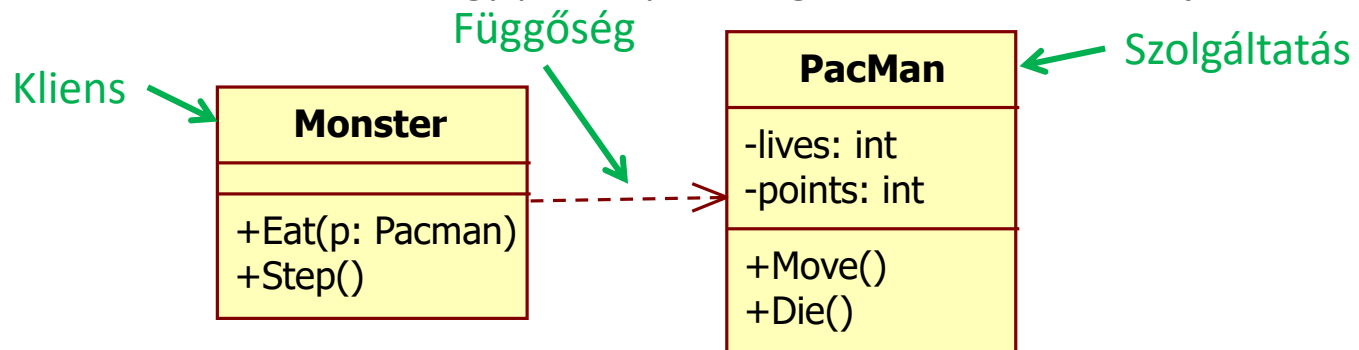
```
class Monster
{
    public void Eat(Pacman* p) { /*...*/ }
    public virtual void Step() = 0;
}
class CleverMonster : public Monster
{
    public void Step() { /*...*/ }
}
```

C# leképezés:

```
public abstract class Monster
{
    public void Eat(Pacman p) { /*...*/ }
    public abstract void Step();
}
public class CleverMonster : Monster
{
    public override void Step() { /*...*/ }
}
```

# Függőség (dependency)

- Egy kliens-szolgáltatás kapcsolatot definiál két modellelem között
- A szolgáltatás megváltozása magával vonhatja a kliens megváltoztatását
- Calssifier-ek között gyenge, ideiglenes kapcsolat:
  - csak egy függvényhívás erejéig tart
  - példák:
    - a kliens paraméterként kap egy példányt a szolgáltatásból
    - a kliens visszaad egy példányt a szolgáltatásból
    - a kliens létrehoz egy példányt a szolgáltatásból
    - a kliens szerez valahonnan egy példányt a szolgáltatásból és használja azt

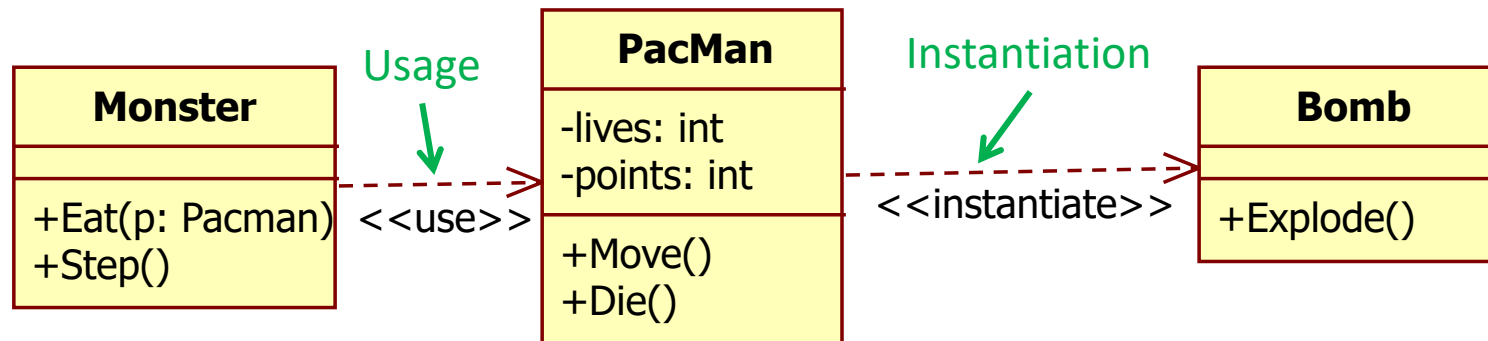


A **Monster** osztály használja a **PacMan** osztályt:  
meghívja a `Die()` metódusát, amikor megeszi (`Eat`) a **PacMan**-t



# Függőség (dependency)

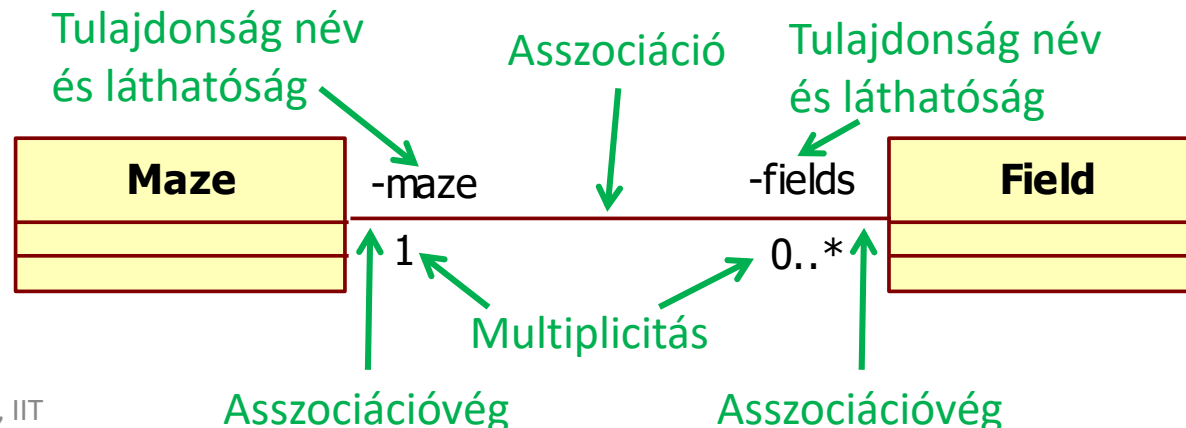
- A függőség jelentése pontosítható sztereotípiával:



- **<<use>>**: a kliensnek szüksége van a szolgáltatásra a működéshez, de a használat pontos módja nincs specifikálva
- **<<instantiate>>**: a kliens a működése során új példányt hoz létre a szolgáltatásból
- Függőségek nemcsak classifier-ek között definiálhatók, hanem más modellelemek között is
  - pl. operáció-osztály, komponens-interfész, stb.

# Asszociáció (association)

- Típussal rendelkező példányok közötti szemantikai kapcsolatot jelent
- Classifier-ek között az asszociáció egy erős, permanens kapcsolatot jelez:
  - túléli a metódushívásokat
  - általában valamilyen attribútumban tárolódik egy referencia a szemben lévő classifier-re
  - interfészeknek nincsenek attribútumaik, de úgy viselkednek, mintha lenne nekik
    - pl. getter-setter függvények Javában, property-k C#-ban, etc.
- Asszociációvég (association end):
  - ez egy tulajdonság (property): van neve, típusa, láthatósága, multiplicitása
  - ha a nevet elhagyjuk, akkor a név tipikusan a classifier neve kisbetűsítve
  - a típus az asszociációvégnél lévő classifier
  - a tulajdonságot a szemközti classifier tárolja



# Asszociáció (association)



C# leképezés:

```
public class Maze {
    private List<Field> fields;
}
public class Field {
    private Maze maze;
}
```

Java leképezés:

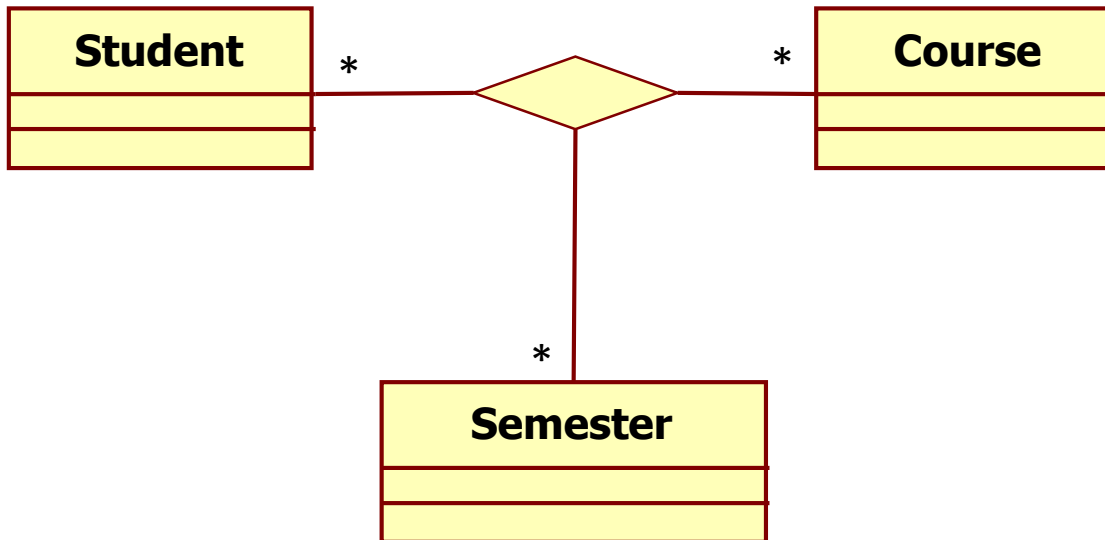
```
public class Maze {
    private ArrayList<Field> fields;
}
public class Field {
    private Maze maze;
}
```

C++ leképezés:

```
class Maze {
private:
    vector<Field*> fields;
}
class Field {
private:
    Maze* maze;
}
```

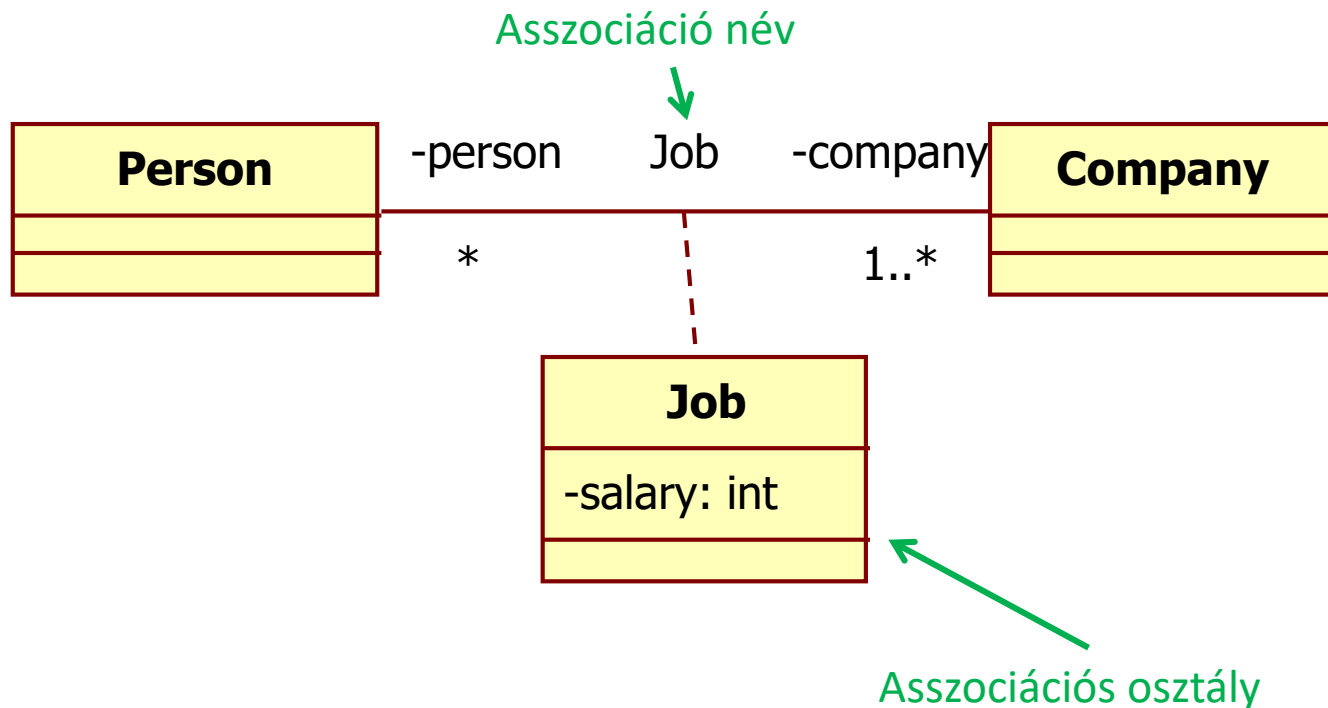
# Többvégű asszociáció

- Az asszociációknak általában 2 végük van: bináris asszociáció (binary association)
- De lehet több is: N-végű asszociáció (N-ary association)



# Asszociációs osztály (association class)

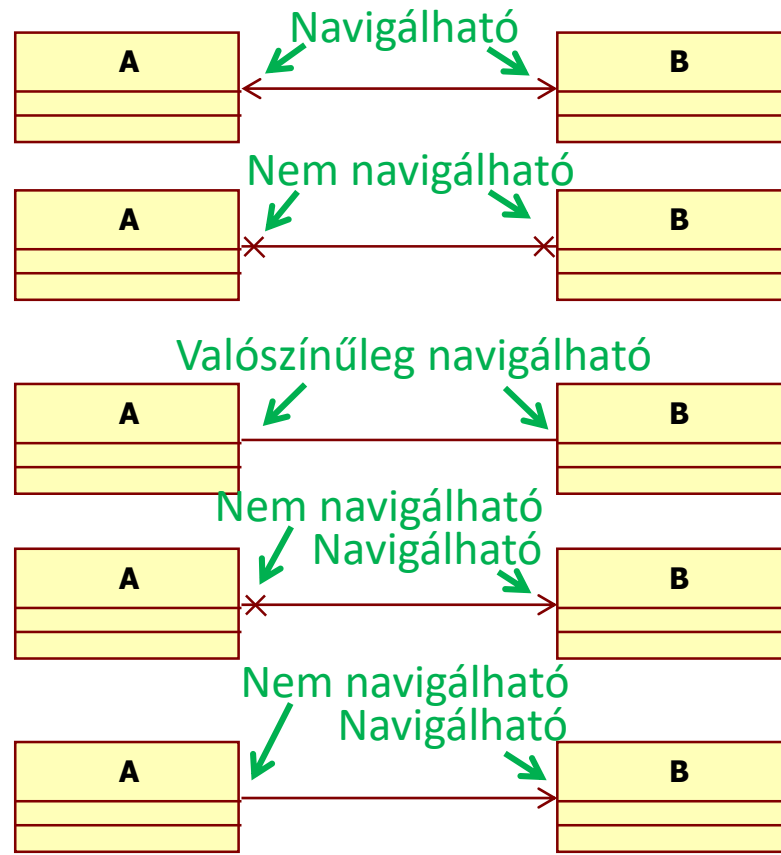
- Egy asszociációnak lehetnek tulajdonságai és operációi
- Egy asszociációs osztály tudja ezeket reprezentálni



(Az asszociáció neve opcionális. Akkor is használható, ha nincs asszociációs osztály.)

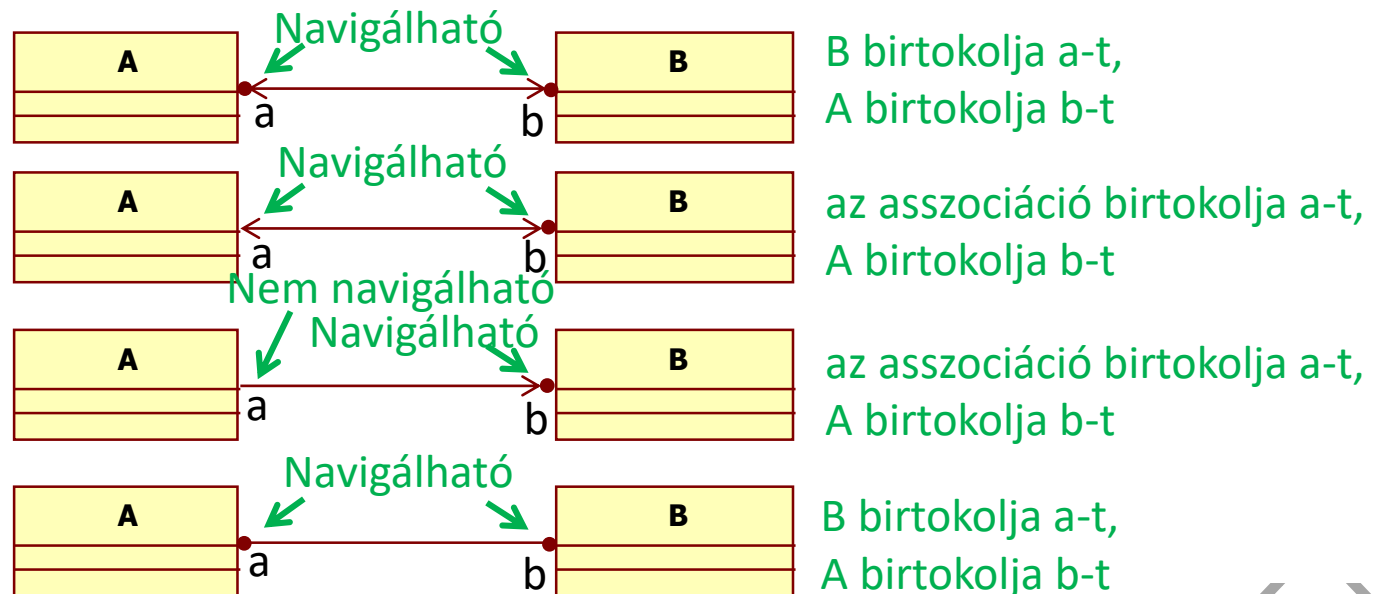
# Navigálhatóság (navigability)

- A navigálhatóság azt jelenti, hogy a példányok hatékonyan elérhetők a másik oldali példányokból
- A hatékonyság pontos jelentése implementációfüggő, de tipikusan direkt referencia/pointer szokott lenni
- Ha egy vég nem navigálható, akkor vagy nincs átjárás, vagy van, de ha van, akkor nem feltétlenül hatékony



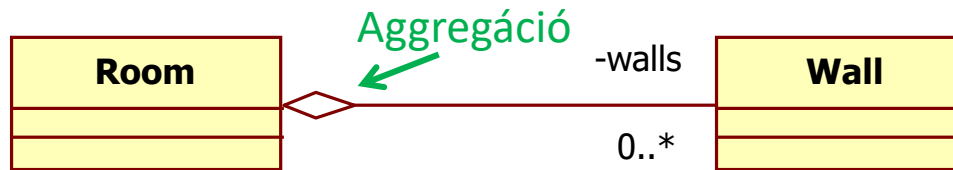
# Birtokolt vég (owned end)

- Egy asszociációvéget reprezentáló tulajdonságot (property) birtokolhatja az asszociáció vagy birtokolhatja a másik végen lévő classifier
- Ha a classifier birtokol: egy pöttyel jelezzük az asszociációvégnél
  - ilyenkor nem kell feltüntetni a tulajdonságot a classifier-en belül
  - egyben navigálhatóságot is jelent
- A birtoklás jelzése nem kötelező: nem biztos, hogy minden modellező eszköz támogatja



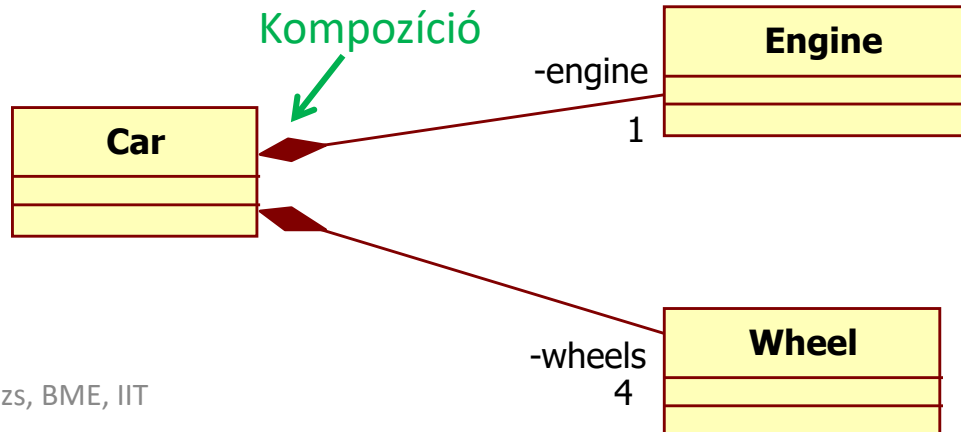
# Aggregáció (aggregation), tartalmazás (composition)

- Az aggregációval azt modellezzük, amikor egy objektum valamilyen objektumokat csoportosít
- **Megosztott aggregáció (shared aggregation):** az aggregáció gyenge formája, amikor az aggregált objektum több csoportosításokban is részt vehet



Egy szobának vannak falai, de egy fal több szobához is tartozhat

- **Kompozit aggregáció (composite aggregation):** az aggregáció erős formája, a tartalmazott objektum egyszerre csak egy csoportosításban szerepelhet
  - Ha a tartalmazó objektumot töröljük/másoljuk, a tartalmazott objektumok is törlődnek/másolódnak
  - Objektumok között a tartalmazásoknak irányított körmentes gráfot kell alkotniuk



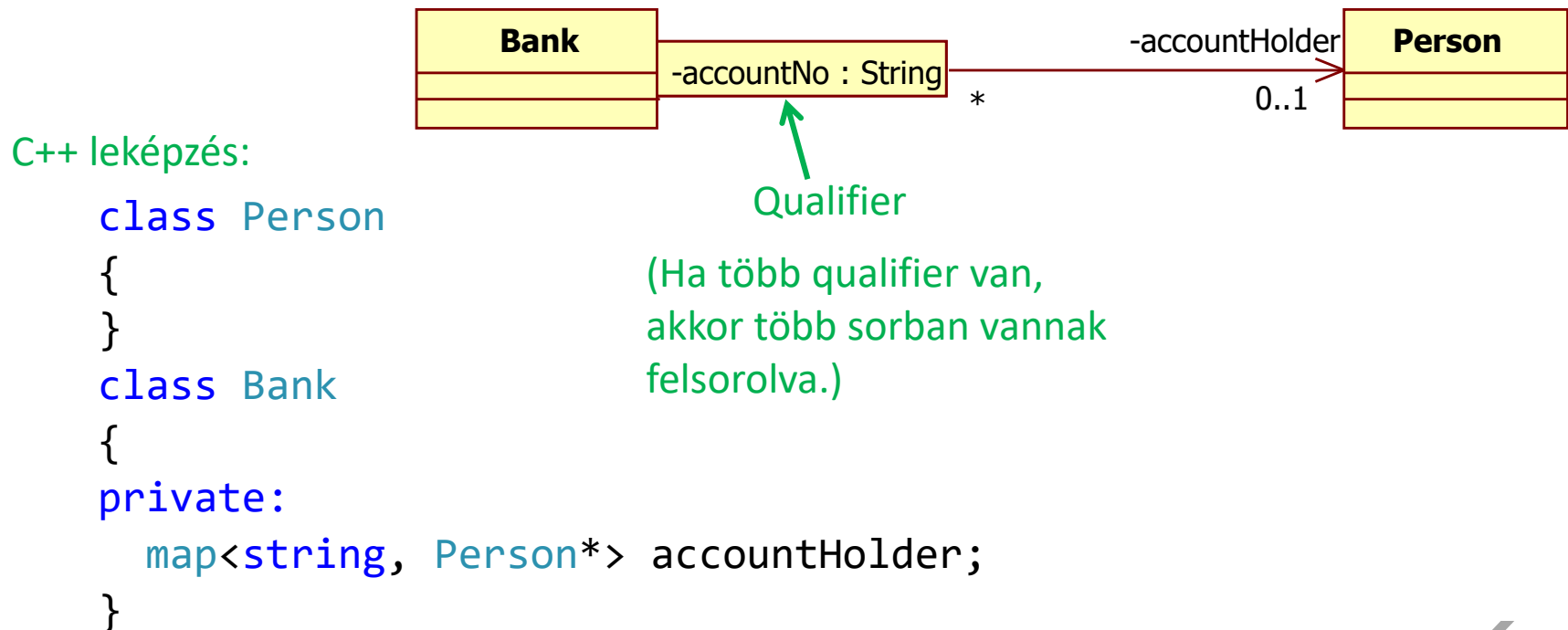
Egy autó tartalmaz egy motort és négy kereket

Ha az autó megsemmisül, a motor és a kerekek is megsemmisülnek.

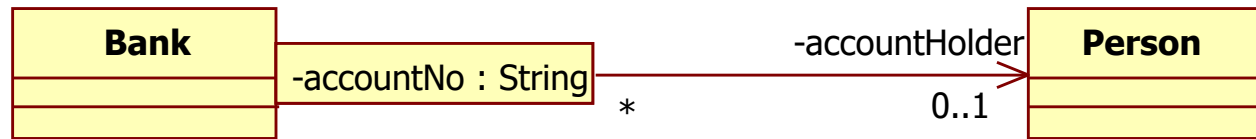


# Minősítő (qualifier)

- Egy minősített asszociációvég partíciókra osztja a másik oldalon lévő objektumokat
- Minden partíciót egy kulcs (qualifier value) jellemez
- A másik oldalon lévő multiplicitás az egyes partíciókban lévő objektumok számát adja meg (nem a partíciók számát!)



# Minősítő (qualifier)



Java leképezés:

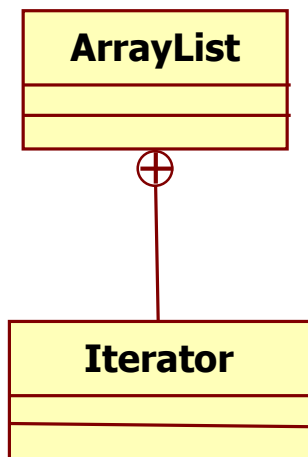
```
public class Person
{
}
public class Bank
{
    private Map<String, Person> accountHolder;
}
```

C# leképezés:

```
public class Person
{
}
public class Bank
{
    private Dictionary<string, Person> accountHolder;
}
```

# Beágyazott osztály (nested class)

- A beágyazott osztályt egy másik osztályon belül definiáljuk



C++ leképezés:

```
class ArrayList
{
public:
    class Iterator
    {
    };
};
```

C# leképezés:

```
public class ArrayList
{
    public class Iterator
    {
    }
}
```

Java leképezés:

```
public class ArrayList
{
    public static class Iterator
    {
    }
}
```

# Tulajdonság módosítók (property modifiers)

- A tulajdonság jelentését pontosítják
  - pl. csak olvasható-e, tárolhatja-e ugyanazt az elemet többször, sorrendben tárolja-e az elemeket, stb.
- Kapcsos zárójelben szerepelnek a tulajdonság után

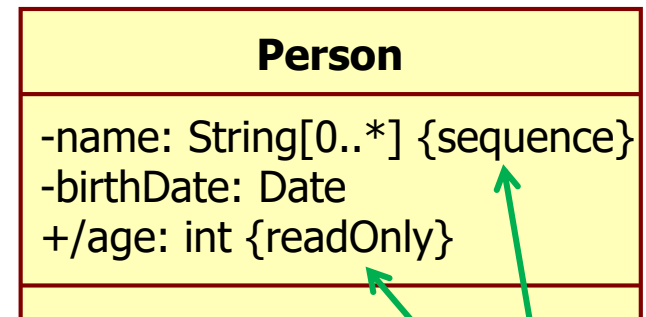
C# leképezés:

```
public class Person
{
    public List<string> Name { get; }
    public DateTime BirthDate { get; set; }
    public int Age { get { /*...*/ } }
    //...
}
```

Java leképezés:

```
public class Person {
    private List<String> name;
    private DateTime birthDate;

    public List<String> getName() { return name; }
    public DateTime getBirthDate() { return birthDate; }
    public void setBirthDate(DateTime value) { birthDate = value; }
    public int getAge() { /*...*/ }
    //...
}
```



Tulajdonság módosító

# Tulajdonság módosítók (property modifiers)

Módosító	Jelentés
readOnly	a tulajdonság csak olvasható
union	a tulajdonság értéke a részhalmazzaiból számolt unió
subsets <i>&lt;propname&gt;</i>	a tulajdonság a <i>&lt;propname&gt;</i> részhalmaza
redefines <i>&lt;propname&gt;</i>	a tulajdonság átdefiniálja az örökölt <i>&lt;propname&gt;</i> tulajdonságot
ordered	az értékek sorrendjét megtartja
unordered	az értékek sorrendjét nem feltétlenül tartja (ez az alapértelmezett)
unique	a több értékű tulajdonságban nincsenek duplikáltan tárolt értékek
nonunique	a több értékű tulajdonságban lehetnek duplikáltan tárolt értékek
sequence (or seq)	a tulajdonság egy lista (nonunique és ordered)
id	a tulajdonság részt vesz az objektumok azonosításában

# Tulajdonság módosítók leképzése kollekciókra

Módosítók	C++ STL	Java	C#
{unique, unordered}	unordered_set	Set interfész, HashSet osztály	ISet interfész, HashSet osztály
{nonunique, unordered}	unordered_multiset	List interfész <sup>1</sup> , ArrayList osztály <sup>1</sup>	ICollection interfész <sup>1</sup> , List osztály <sup>1</sup>
{unique, ordered}	vector <sup>2</sup>	LinkedHashSet osztály, List interfész <sup>2</sup> , ArrayList osztály <sup>2</sup>	ICollection interfész <sup>2</sup> , List osztály <sup>2</sup>
{nonunique, ordered} vagy {sequence} vagy {seq}	vector	List interfész, ArrayList osztály	ICollection interfész, List osztály

Egyedi (unique) kollekciók egy értéket csak egyszer tárolhatnak.

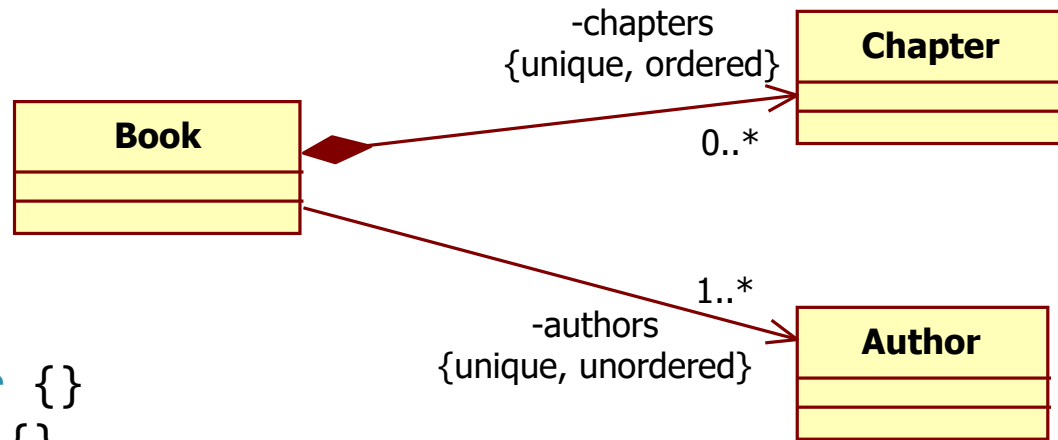
Rendezett (ordered) kollekciók megtartják a beszúrás sorrendjét és az elemeik tipikusan indexelhetők.

<sup>1</sup> Nem használjuk ki a rendezettséget

Dr. Simon Balázs, BME, IIT

<sup>2</sup> Az elemek egyediségét nekünk kell biztosítani

# Tulajdonság módosítók példa



## C# leképezés:

```
public class Chapter {}
public class Author {}
public class Book
{
    private List<Chapter> chapters;
    private HashSet<Author> authors;
}
```

## Java leképezés:

```
public class Chapter {}
public class Author {}
public class Book {
    private ArrayList<Chapter> chapters;
    private HashSet<Author> authors;
}
```

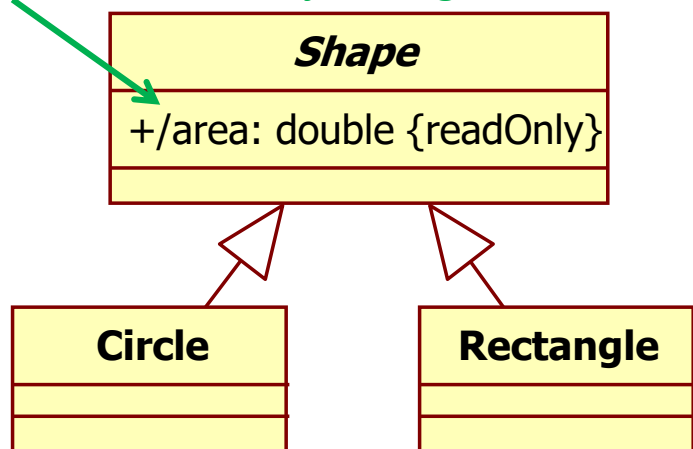
## C++ leképezés:

```
class Chapter {};
class Author {};
class Book
{
private:
    vector<Chapter*> chapters;
    set<Author*> authors;
};
```

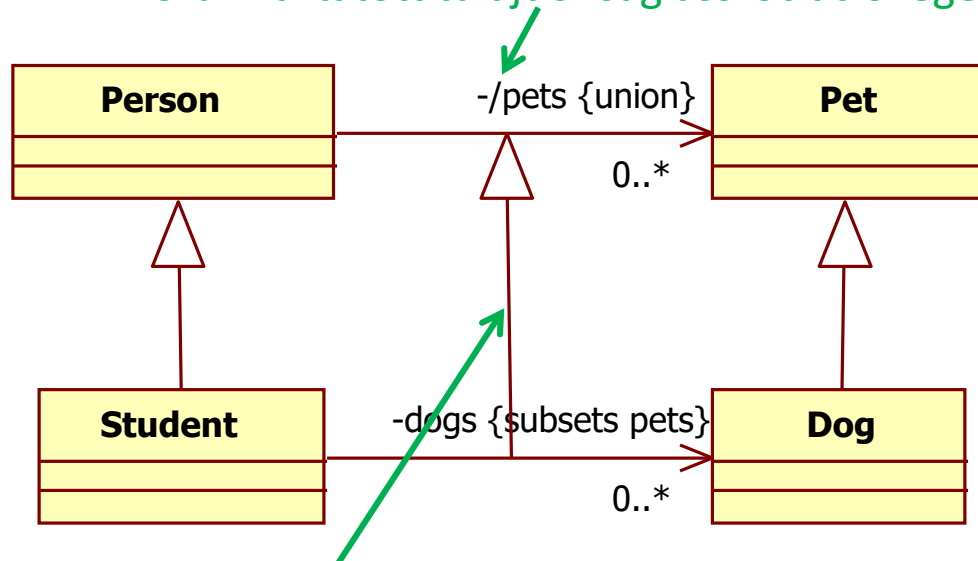
# Származtatott tulajdonságok (derived properties)

- A származtatott tulajdonságok értéke valamilyen számítás eredménye
- Gyakran csak olvashatók is
- Ha mégis írható, akkor az implementációtól elvárt, hogy a szükséges egyéb értékadásokat is elvégezze (pl. olyan más tulajdonságokban, amelyekből ennek a tulajdonságnak az értéke számítódik)
- Jelölés: a név előtti perjel (/)
- Példák:

Származtatott tulajdonság



Származtatott tulajdonság asszociációvén



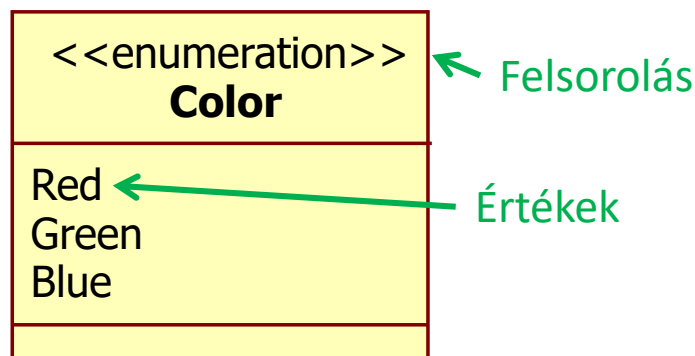
(Asszociációk között is értelmezett az öröklődés.)

(Asszociációk is lehetnek származtatottak. Jelölés: asszociációnév előtti perjel.)



# Felsorolás (enumeration)

- A felsorolás adattípus értékei a modellben felsorolt fix értékek (enumeration literals)



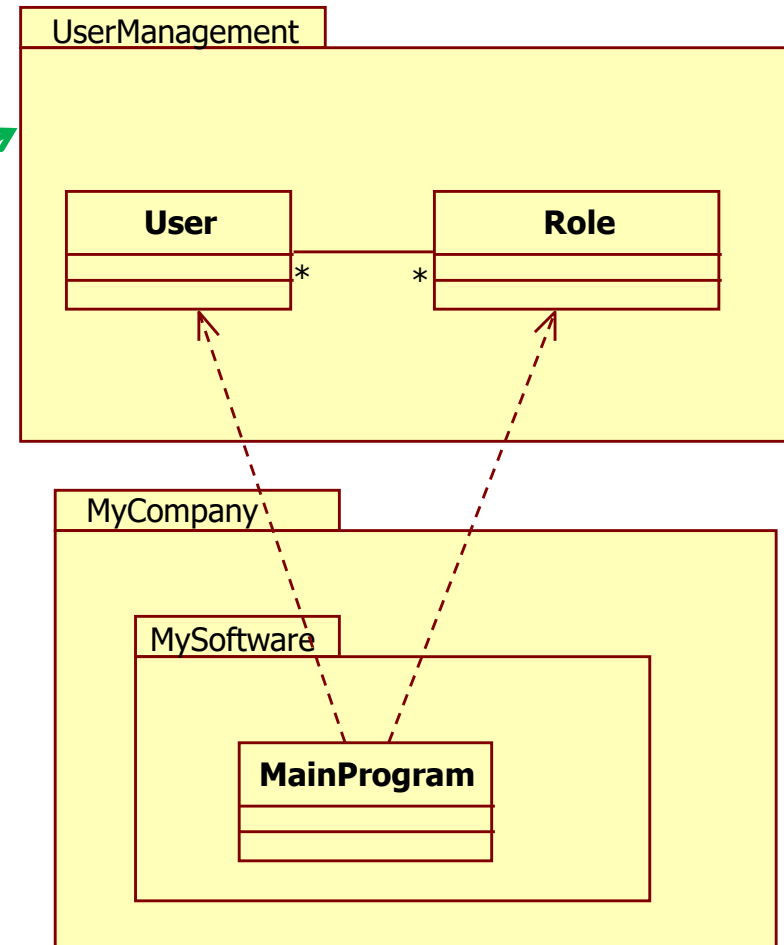
C++/Java/C# leképzés:

```
enum Color
{
    Red,
    Green,
    Blue
}
```

# Csomag (package)

- **Csomag:** különböző elemek csoportosítására szolgál
- Tipikus programnyelvi leképzés: csomag vagy névter

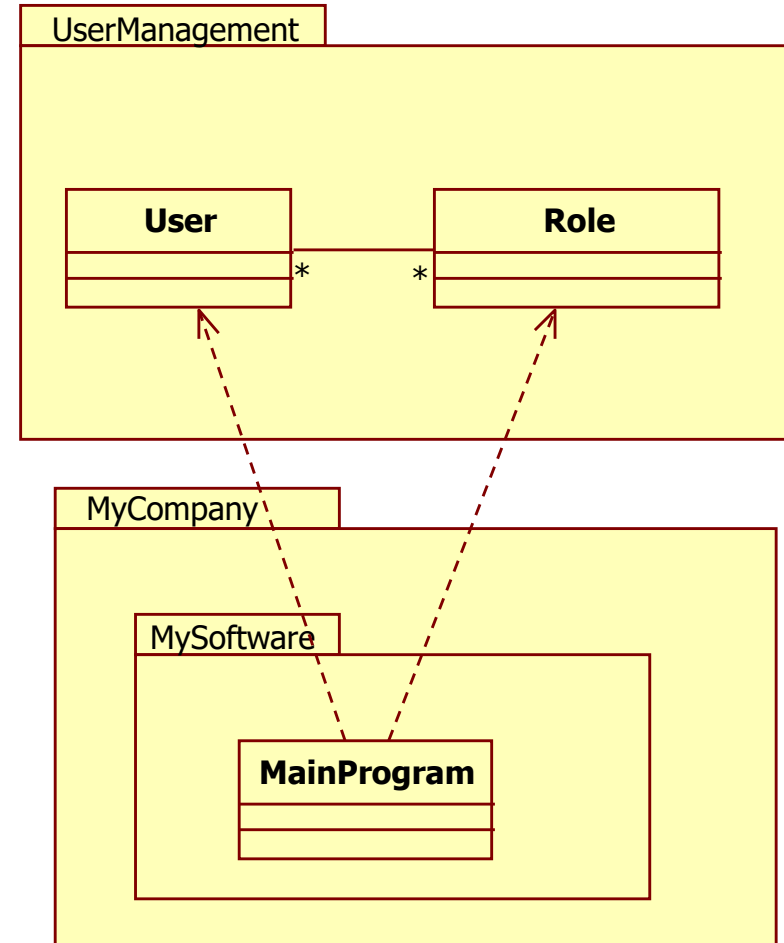
Csomag



# Csomag (package)

C++ leképezés:

```
namespace UserManagement
{
    class User
    {
        set<Role*> roles;
    };
    class Role
    {
        set<User*> users;
    };
}
namespace MyCompany
{
    namespace MySoftware
    {
        using namespace UserManagement;
        class MainProgram
        {
        };
    }
}
```



# Csomag (package)

Java leképzés:

User.java:

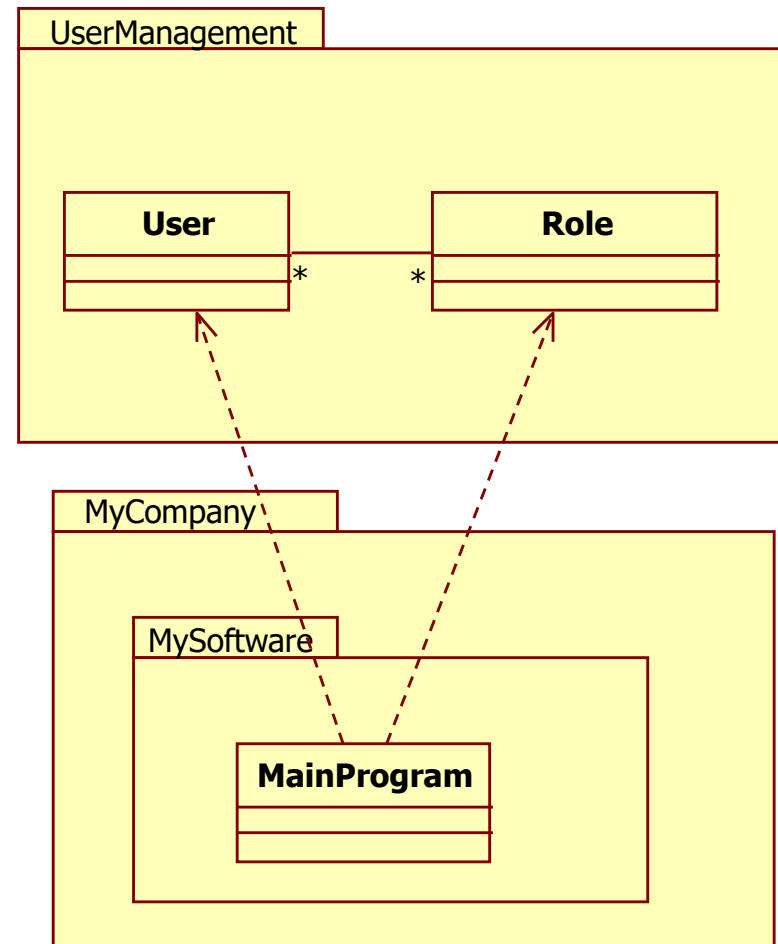
```
package usermanagement;  
import java.util.HashSet;  
public class User {  
    private HashSet<Role> roles;  
}
```

Role.java:

```
package usermanagement;  
import java.util.HashSet;  
public class Role {  
    private HashSet<User> users;  
}
```

MainProgram.java:

```
package mycompany.mysoftware;  
import usermanagement.User;  
import usermanagement.Role;  
public class MainProgram {  
}
```



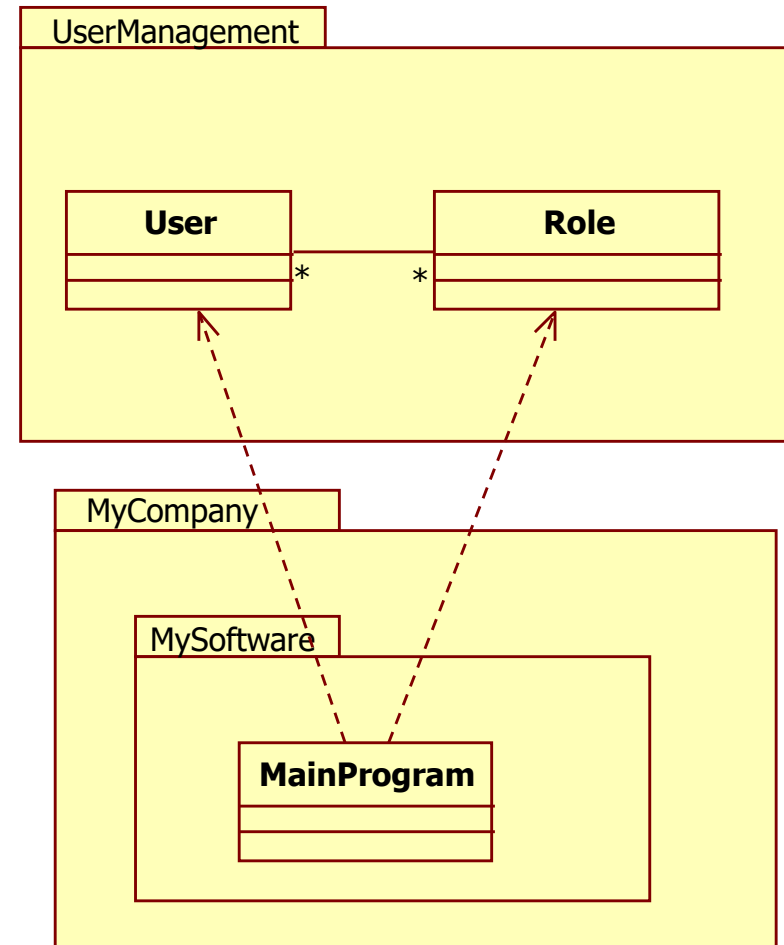
# Csomag (package)

C# leképezés:

```
namespace UserManagement
{
    public class User
    {
        private HashSet<Role> roles;
    }
    public class Role
    {
        private HashSet<User> users;
    }
}

namespace MyCompany.MySoftware
{
    using UserManagement;

    public class MainProgram
    {
    }
}
```



# Hol tartunk?

---

## Strukturális UML diagrammok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildíagram	

## Viselkedési UML diagrammok:

Use case diagram	Aktivitásdiagram	Szekvenciadiagram	Kommunikációs diagram
Állapotdiagram	Időzítődiagram	Interakciós áttekintő diagram	

# 00 fogalmak

---

## ■ Absztrakció (abstraction):

- az adott kontextusban felesleges részletek elhanyagolása
- a világ objektumai leképezhetők objektumokra a programban

## ■ Osztályozás (classification):

- közös tulajdonságokkal és közös viselkedéssel bíró dolgok csoportosítása
- a közös tulajdonságokat és a közös viselkedést az osztály írja le

## ■ Egységbezárás (encapsulation):

- egy osztálynak nem szabad engednie, hogy kívülről közvetlenül hozzáférjenek a mezőikhez
- csak metódusokon keresztül szabad
- a mezőknek privátnak kell lenniük

## ■ Öröklődés (inheritance):

- a leszármazott osztály újrahasznosítja az ős viselkedését
- az öröklődés “az-egy” kapcsolat a leszármazott és az ős között
- fontos:
  - az öröklődést csak a viselkedés újrahasznosítására használjuk!
  - soha ne használjunk öröklődést az adatok újrahasznosítására! (helyette: delegáció)

## ■ Polimorfizmus (polymorphism):

- a hívónak ne kelljen törődnie azzal, hogy egy objektum típusa az ős vagy annak valamelyik leszármazottja
- megvalósítása: virtuális függvények és azok felüldefiniálása



## ■ Csatolás (coupling):

- az egyes modulok/komponensek/osztályok/függvények közötti függőség mértéke
- a közöttük lévő kapcsolat erőssége
- a *laza csatolás (low coupling)* előnyös a karbantarthatóság szempontjából: egy változás csak kis mértékben hat ki a rendszer más részeire

## ■ Kohézió (cohesion):

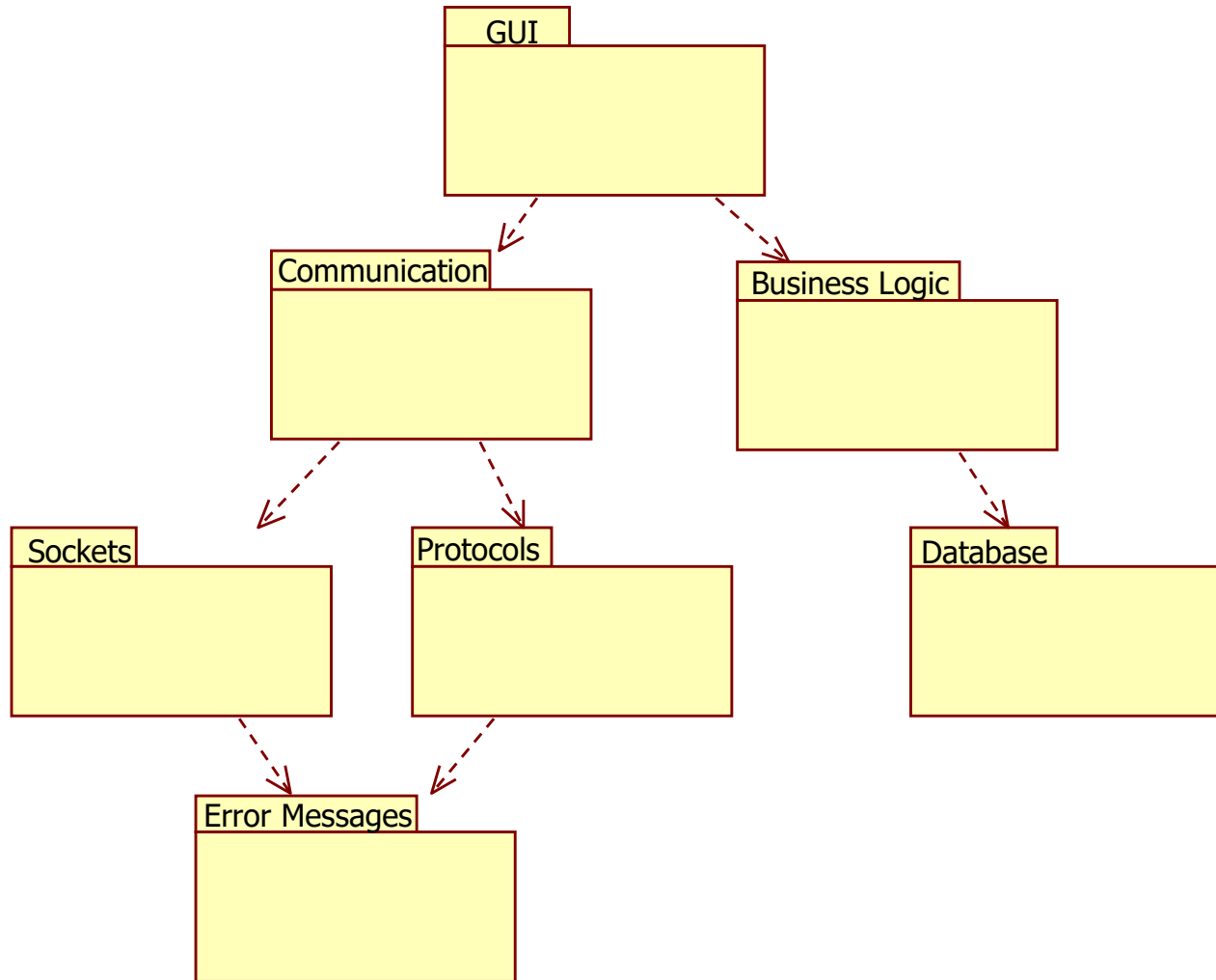
- annak a mértéke, hogy a modulok/komponensek/osztályok elemei mennyire tartoznak össze
- a bennük lévő elemek közötti kapcsolat erőssége
- az *erős kohézió (high cohesion)* előnyös a karbantarthatóság szempontjából: az összetartozó funkciók egy helyen vannak lokalizálva

# Csomagdiagram (Package Diagram)

---

# Csomagdiagram (Package Diagram)

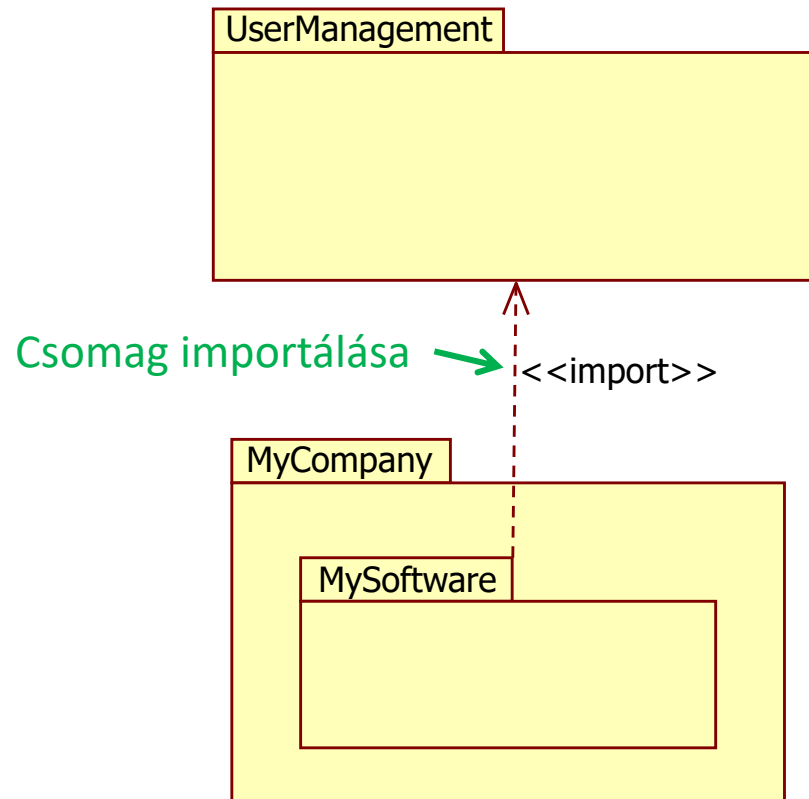
- A csomagdiagram csomagok közötti **függőségeket (dependencies)** ábrázol
- Példa:



- Két speciális függőség van: **import** és **merge**

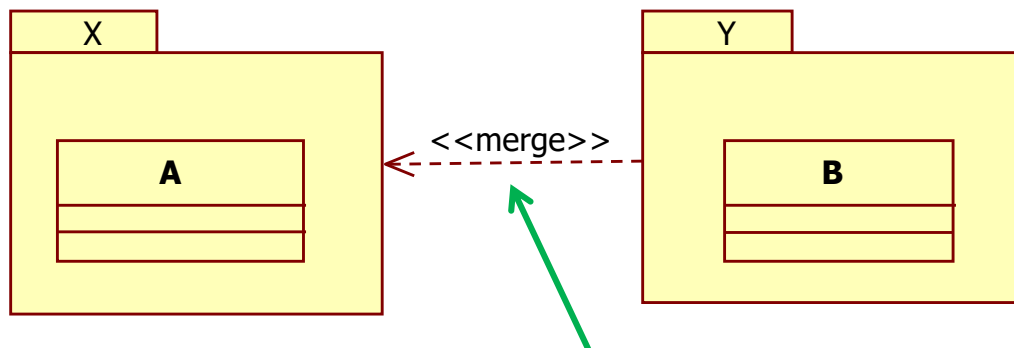
# Csomag importálása (Package import)

- Az importáló csomag a saját névterén belül elérhetővé teszi az importált csomag elemeit
  - programnyelvekben: import/include/using



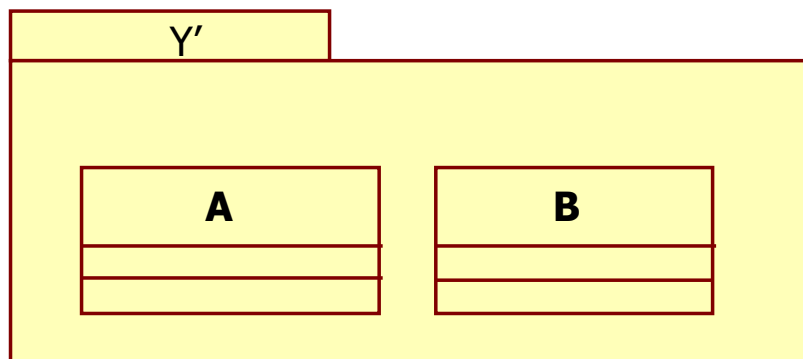
# Csomagok összefésülése (Package merge)

- Irányított kapcsolat két csomag között: azt jelzi hogy a két csomag tartalmát egyesíteni kell
  - nem képezhető le programnyelvekre
  - a kombinációs szabályok nagyon bonyolultak (ld. a szabványt)



Csomagok összefésülése: X-et Y-ba kell fésülni

- A keletkező csomag egy új csomag:



# Hol tartunk?

---

## Strukturális UML diagrammok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildíagram	

## Viselkedési UML diagrammok:

Use case diagram	Aktivitásdiagram	Szekvenciadiagram	Kommunikációs diagram
Állapotdiagram	Időzítődiagram	Interakciós áttekintő diagram	

# Objektumdiagram (Object Diagram)

---

# Objektumdiagram (Object Diagram)

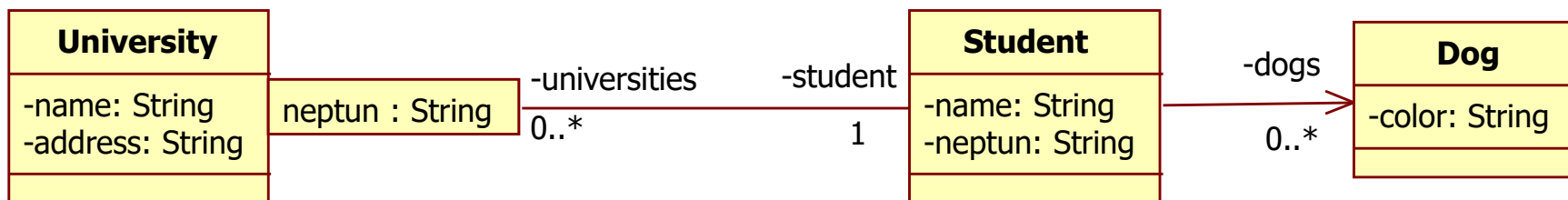
---

- Az objektumdiagram egy példányokból álló gráf (a csomópontok objektumok és értékek)
- Az objektumdiagram az osztálydiagram egy példánya:
  - osztályok példányai: **objektum (object)** vagy **példány specifikáció (instance specification)**
  - asszociációk példányai: **link**
- A rendszer részletes állapotáról ad egy képet egy adott időpontban
  - Ne keverjük össze az objektumdiagramon szereplő elemeket azokkal a szoftver memóriájában létező dinamikus példányokkal, amelyeket modelleznek!
  - különböző időpontokban különböző objektumdiagramok ábrázolhatják ugyanazokat a dinamikus példányokat
- Az objektumdiagramok használata szűkterű: csak példákat adnak a megfelelő adatstruktúrákra

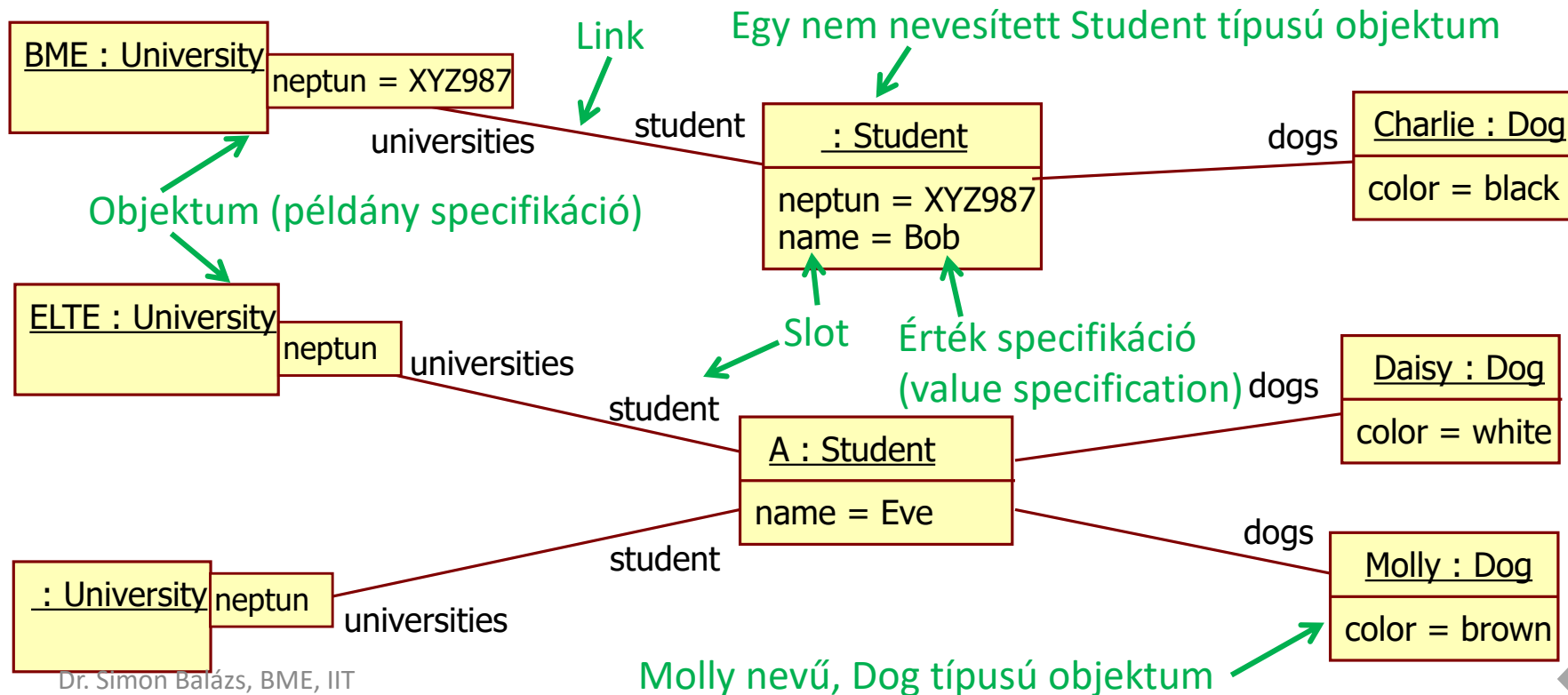


# Objektumdiagram példa

## Osztálydiagram:



## Egy lehetséges objektumdiagram a fenti osztálydiagramhoz:



# Hol tartunk?

---

## Strukturális UML diagrammok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildíagram	

## Viselkedési UML diagrammok:

Use case diagram	Aktivitásdiagram	Szekvenciadiagram	Kommunikációs diagram
Állapotdiagram	Időzítődiagram	Interakciós áttekintő diagram	

# Összefoglalás

---

# Összefoglalás

---

- UML diagrammok:
  - osztálydiagram
  - csomagdiagram
  - objektumdiagram

# Hol tartunk?

---

## Strukturális UML diagrammok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildíagram	

## Viselkedési UML diagrammok:

Use case diagram	Aktivitásdiagram	Szekvenciadiagram	Kommunikációs diagram
Állapotdiagram	Időzítődiagram	Interakciós áttekintő diagram	