Unified Modeling Language

Szoftvertechnológia

Dr. Goldschmidt Balázs

BME, IIT

Tartalom

- Modellezés
- Unified Modeling Language (UML)
- UML diagramok:
  - Use Case diagram
  - Aktivitásdiagram
  - Komponensdiagram
  - Telepítési diagram

Modellezés

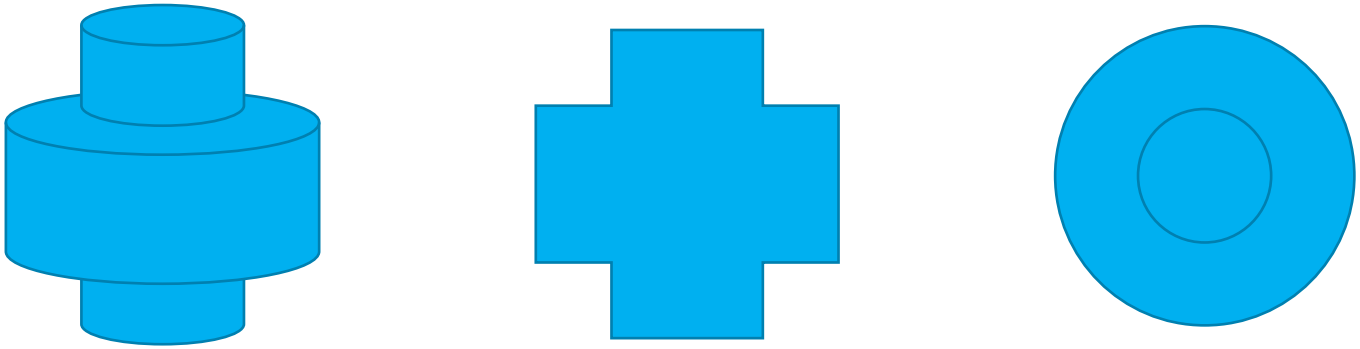
- Egy modell a valóság egy egyszerűsített változata
- Egy jó modell hangsúlyozza a fontos részleteket és elhanyagolja az irreleváns részleteket
- Példa: Budapest metrótérkép
  - fontos részletek:
    - állomások nevei
    - állomások sorrendje
    - átszállási pontok
  - irreleváns részletek:
    - a város utcái
    - állomások közötti távolság
    - az alagút szerkezeti felépítése

Egyéb modellezési példák

- A Naprendszer modellje:
  - fontos részletek: tömeg, sebesség, bolygók távolsága
  - irreleváns részletek: a bolygókat alkotó atomok
- Egy repülőgép modellje a szélcsatornában:
  - fontos részletek: alak, aerodinamika
  - irreleváns részletek: ülőhelyek száma, felszolgált étel, pilóta
- Egy repülőgép modellje helyfoglaláshoz:
  - fontos részletek: ülőhelyek száma, felszolgált étel
  - irreleváns részletek: alak, aerodinamika, pilóta

Nézetek

- A fontos és az irreleváns részletek függnek attól, hogy:
  - mire használjuk a modellt
  - ki fogja olvasni a modellt
- Minél több információt kell a modellnek lefednie, annál komplexebbé válik
- Néha ugyanaz a modell többféle célra is felhasználható és többféle célközönsége van
  - ilyenkor többféle nézetét is elkészíthetjük ugyanannak a modellnek
  - minden nézet az adott célnak és adott célközönségnek biztosítja a releváns részleteket
- Példa:
  - ugyanaz a modell különböző nézetekben:



UML

- Unified Modeling Language
- Szabvány szoftverrendszerek modellezésére
  - az Object Management Group (OMG) által kiadott szabvány
- Az UML szabvány többféle diagramtípust definiál:
  - mindegyik diagram a rendszer egyfajta nézetét adja
- Az UML szabvány két dolgot ír le:
  - **szintaxis**: hogyan néznek ki az egyes diagramok (nézetek)
  - **szemantika**: mit jelentenek a diagramok egyes elemei
- Mind a szintaxis, mind pedig a szemantika fontos a modell megértéséhez
- A különböző diagramokból kiolvasható szemantikának konzisztensnek kell lennie a diagramok között, különben a modell ellentmondásos és használhatatlan

UML szemantika

- Egy UML modellnek két fontos aspektusa van:
  - **Strukturális szemantika (statikus szemantika)**: a modellezett rendszer egyes elemeinek jelentését definiálja egy adott időpillanatban
  - **Viselkedési szemantika (dinamikus szemantika)**: a modellezett rendszer elemei jelentésének időbeli változását definiálja
- A különböző diagramtípusok a modell különböző aspektusait írják le

UML diagramok típusai

Strukturális UML diagramok:

Component diagram	Deployment diagram	Class diagram	Package diagram
Object diagram	Composite structure diagram	Profile diagram	

Viselkedési UML diagramok:

Use case diagram	Activity diagram	Sequence diagram	Communication diagram
State diagram	Timing diagram	Interaction overview diagram	

UML diagramok típusai – magyarul

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

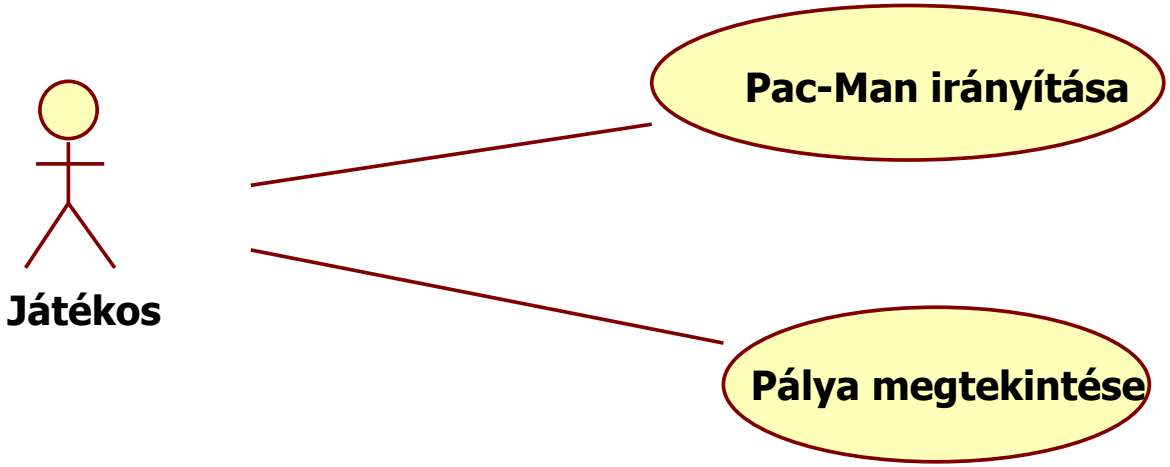
Viselkedési UML diagramok:

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

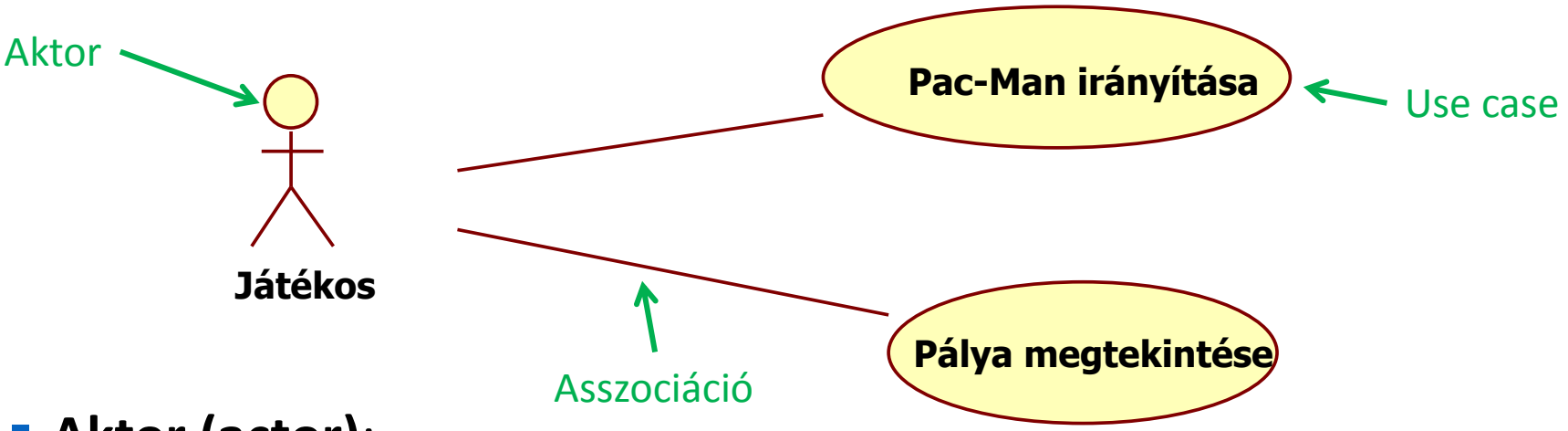
Use Case Diagram

Use Case Diagram

- A rendszer *funkcionális* követelményeit reprezentálja
  - mit kellene a rendszernek csinálnia, hogyan fogják használni a rendszert
  - modellezi a rendszer felhasználóit és a rendszer határait
  - elég egyszerű ahhoz, hogy a megrendelő is megértse
- Példa: Pac-Man játék



Use Case Diagram

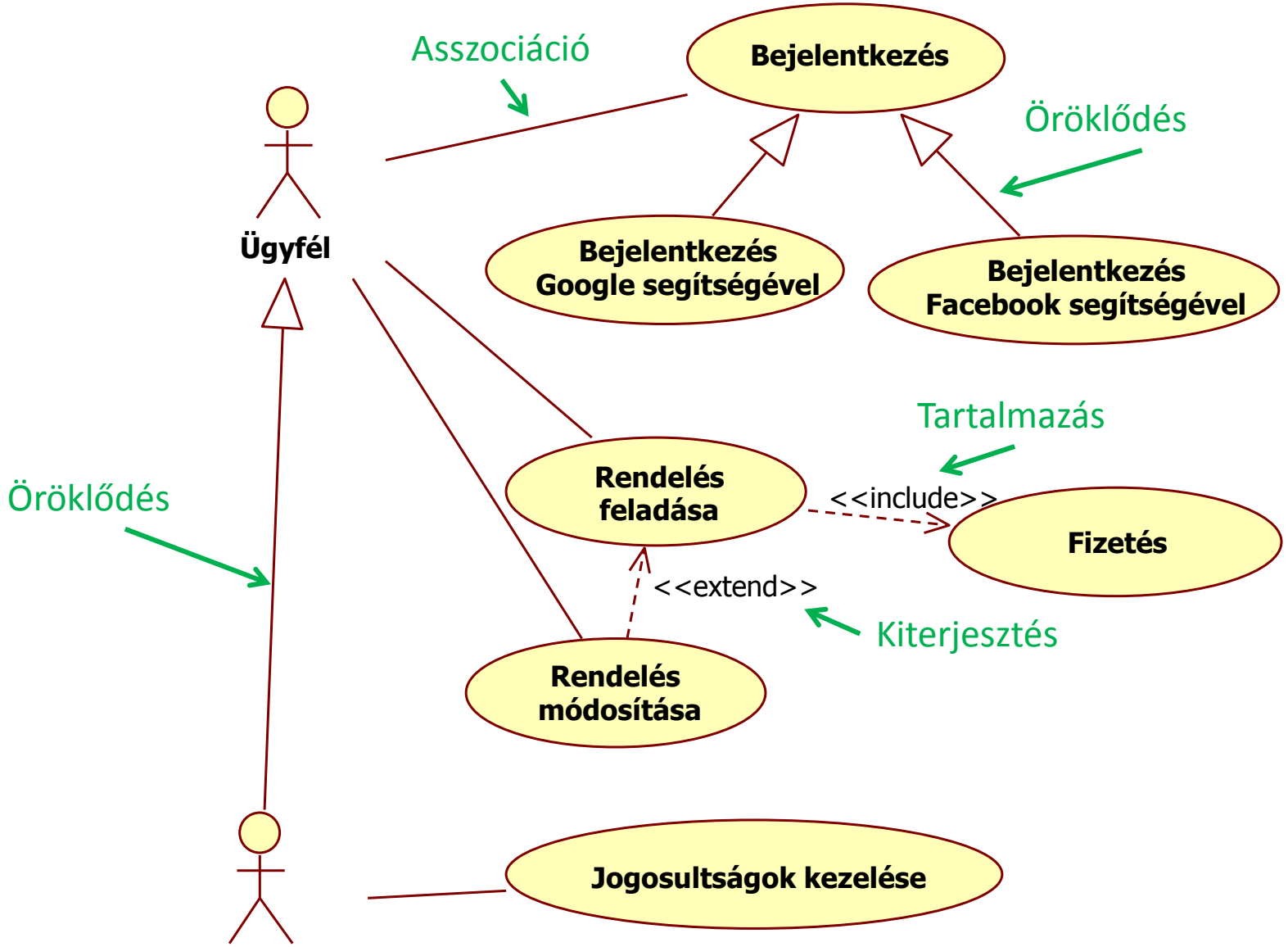


- **Aktor (actor)**:
  - egy felhasználó szerepköre, aki a rendszerrel interakcióba lép
  - lehet ember vagy külső rendszer is
  - (egy fizikai ember többféle szerepkörben is használhatja a rendszert)
- **Használati eset (use case)**:
  - a felhasználó és a rendszer közötti interakciót reprezentálja a felhasználó szemszögéből
  - akciók és interakciók sorozata az aktorok és a rendszer között
- **Asszociáció (association)**:
  - összekapcsolja az aktort azzal a use case-t, amelyikben részt vesz

Use Case Diagram: Kapcsolatok

- Aktor és use case között:
  - **Asszociáció (association)**:
    - összekapcsolja az aktort azzal a use case-t, amelyikben részt vesz
- Aktorok között:
  - **Öröklődés (generalization)**:
    - a leszármazott aktor az ős aktor speciális változata
    - a leszármazott aktor mindazon use case-eket tudja, amiket az ős is
- Use case-ek között:
  - **Öröklődés (generalization)**:
    - a leszármazott use case az ős use case speciális változata
    - a leszármazott use case *pontosítja* az ős use case működését
  - **<<extend>>**:
    - a kiterjesztő use case néhány egyéb lépéssel bővíti a kiterjesztett use case lépéseit meghatározott *kiterjesztési pontokon*
  - **<<include>>**:
    - a tartalmazó use case egy ponton „meghívja” a tartalmazott use case-t

Példák kapcsolatokra



Use case leírás sablon (nem része az UML-nek)

- **Cím:** a use case célja
  - tipikusan: ige + főnév
- **Aktorok:** a use case szereplői
- **Főforgatókönyv:** számozott lépések sorozata
  - lépés: <egy egyszerű, az aktor és a rendszer közötti interakciót leíró kijelentés>
- **Alternatív forgatókönyvek:** külön számozott listák, alternatívánként külön lista
  - alternatíva: <egy feltétel, amely a főforgatókönyvtől eltérő interakciót eredményez>
  - pl. a főforgatókönyv 7-es lépésétől elágazó alternatíva száma 7.A.

Use case leírás példa: Pénzfelvétel use case

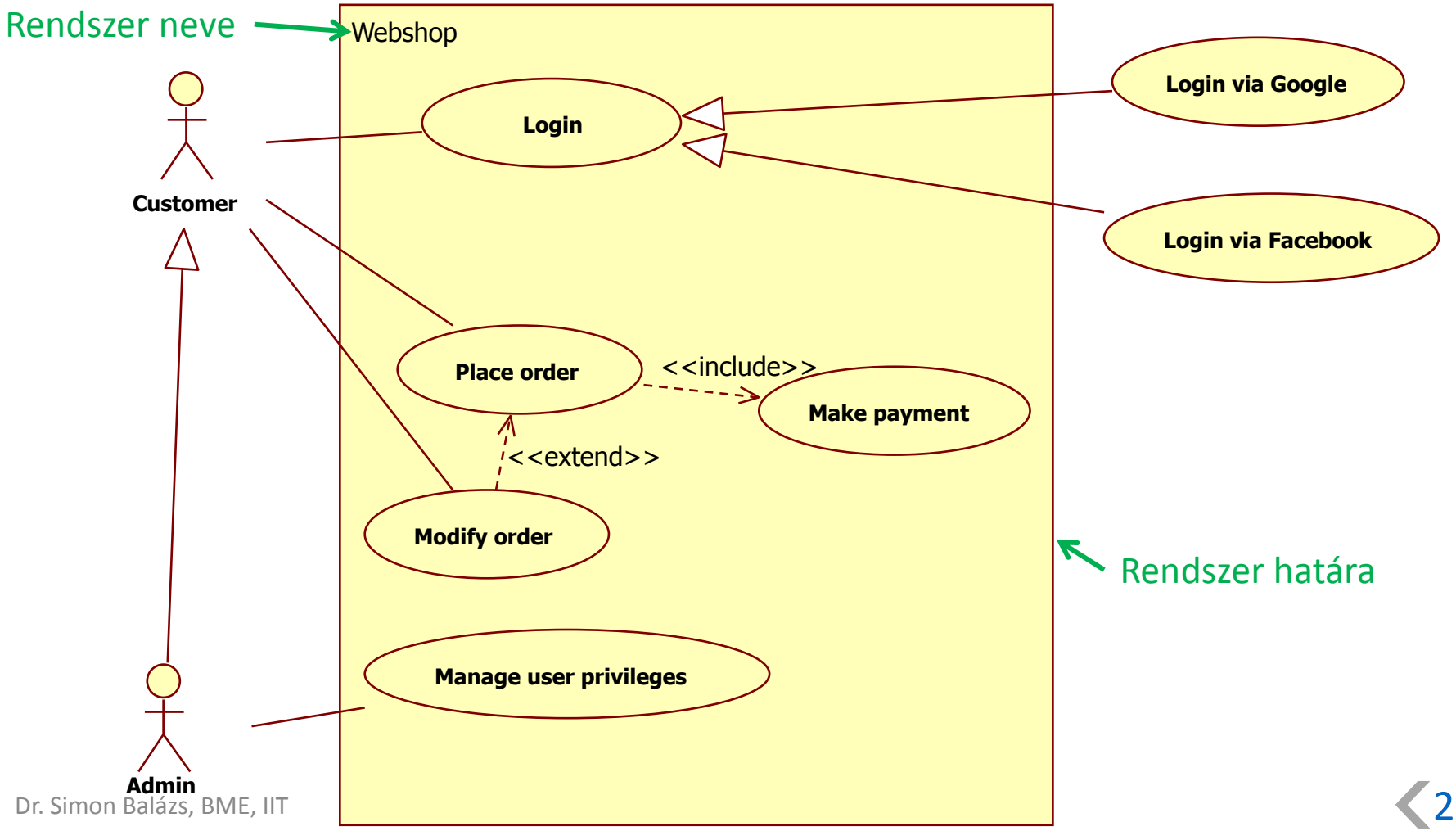
- **Use case:** Pénzfelvétel
- **Aktorok:** Ügyfél, Bank
- **Főforgatókönyv:**
  - 1. Az Ügyfél átadja a bankkártyát és a pinkódot
  - 2. Az ATM ellenőrzi, hogy a bankkártyához tartozik-e a pinkód
  - 3. Az Ügyfél megadja, hogy mennyi pénzt venne fel
  - 4. Az ATM megkérdezi a Bank-ot, hogy ez így rendben van-e
  - 5. A Bank megerősíti, hogy mehet a tranzakció
  - 6. Az ATM kiadja a bankkártyát
  - 7. Az Ügyfél elveszi a bankkártyát
  - 8. Az ATM kinyomtatja a bizonylatot és kiadja a pénzzel együtt
  - 9. Az Ügyfél elveszi a pénzt és a bizonylatot
- **Alternatív forgatókönyv 5.A:**
  - 5.A.1. A Bank jelzi, hogy az Ügyfél számláján nincs elég pénz
  - 5.A.2. Az ATM visszaadja a bankkártyát és hibaüzenetet ír
  - 5.A.3. Az Ügyfél elveszi a bankkártyát

Részletesebb use case leírás

- A use case tartalmának leírására nincs szabvány
- Az előző diákon szereplő minták a leggyakoribb formái a use case-ek leírásának
- De sokkal több mező is lehet egy use case leírásban (Cockburn):
  - Cím, Elsődleges aktor, Cél, Határok, Szint, Megrendelők és érdekeik, Előfeltételek, Utófeltételek (minimális garanciák, sikerességi garanciák), Trigger, Főforgatókönyv, Kiterjesztések, Technológia
- Egyéb lehetséges mezők:
  - Egyedi azonosító, Használat gyakorisága, Kivételes esetek, Lefedett követelmények

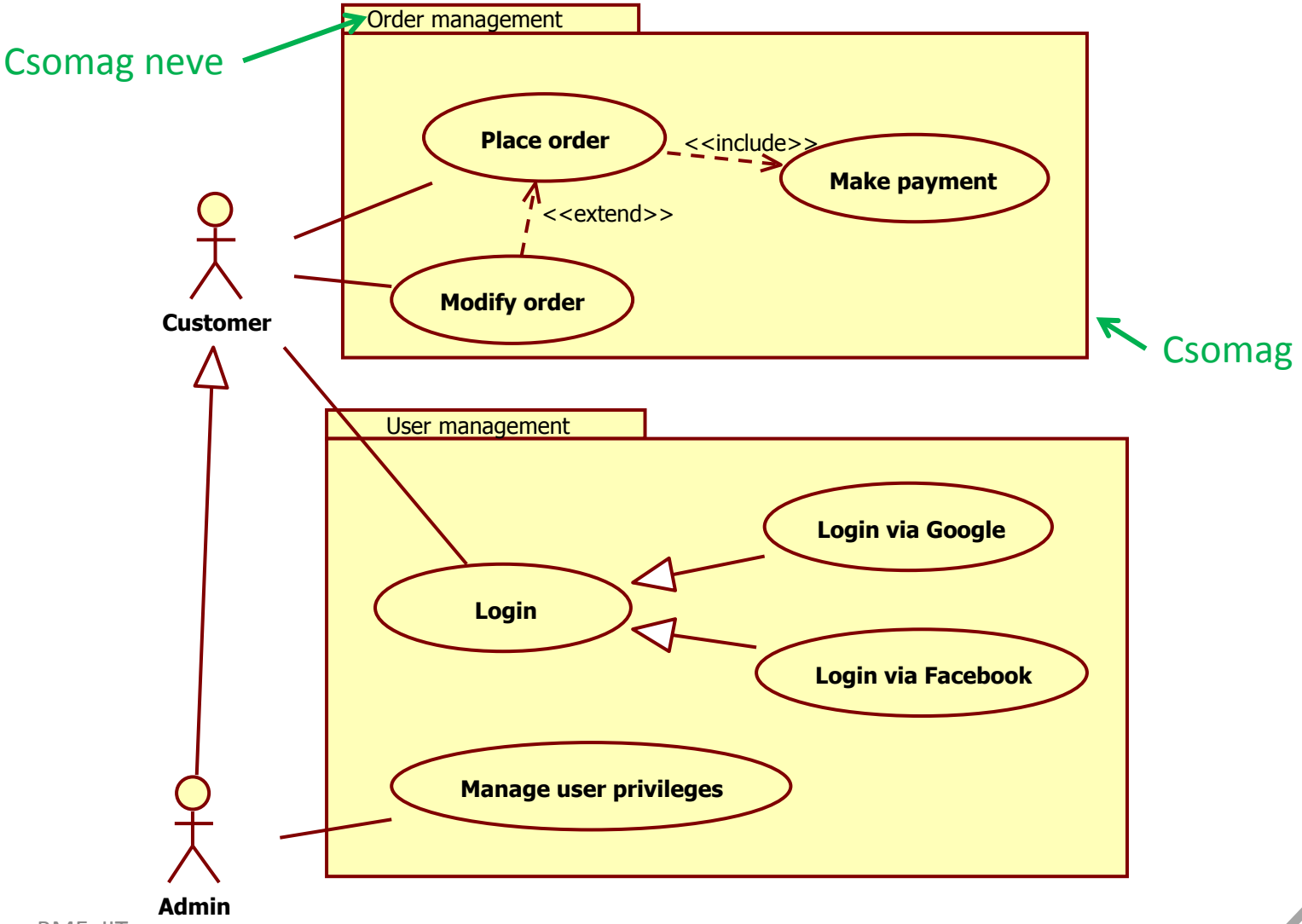
Use Case Diagram: Rendszer határa (opcionális)

- **Rendszer határa (System boundary box)**:
  - keret, amely a rendszer határát jelzi
  - a kereten kívüli use case-ek nem részei a rendszernek



Use Case Diagram: Csomag (opcionális)

- **Csomag (package)**: célja különböző elemek csoportosítása



UML diagramok típusai

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

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

Aktivitásdiagram (Activity Diagram)

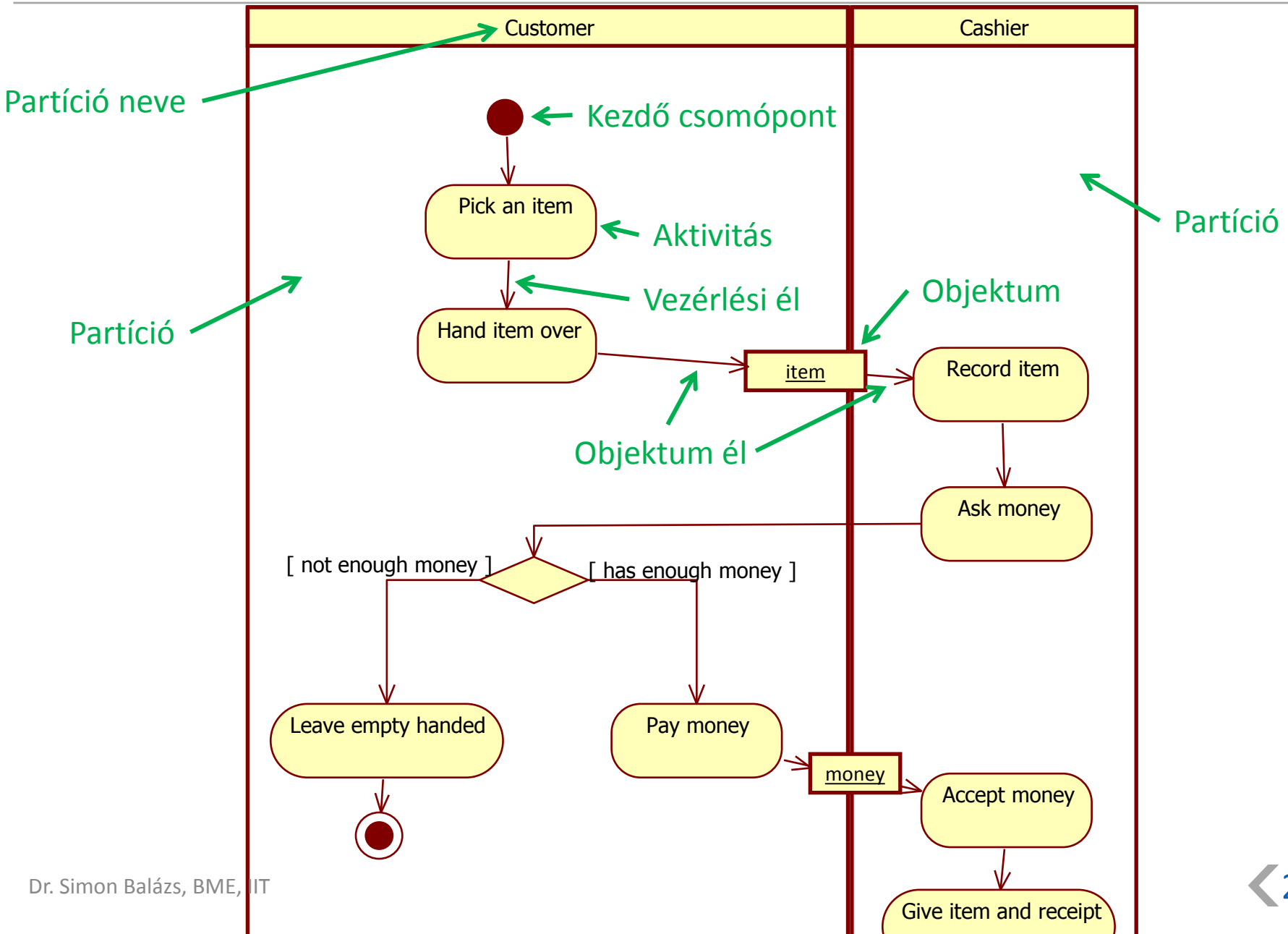
Aktivitásdiagram

- Az aktivitásdiagramok munkafolyamatok lépéseit ábrázolják grafikusan
  - tevékenységek, döntések, ismétlések, párhuzamos működés
- Tipikusan a következők modellezésére használjuk őket:
  - egy adott use case leírása
  - egy a felhasználók és a rendszer közötti üzleti folyamat vagy munkafolyamat
  - egy alkgoritmus lépései
- Az aktivitásdiagramok a *funkcionális* követelmények munkafolyamatként történő formális leírását adják
- Az aktivitásdiagramok többszintűek lehetnek
  - pl. először egy magas absztrakciós szintű folyamat, majd később egy alacsonyabb absztrakciós szintű folyamat a lépések finomításával

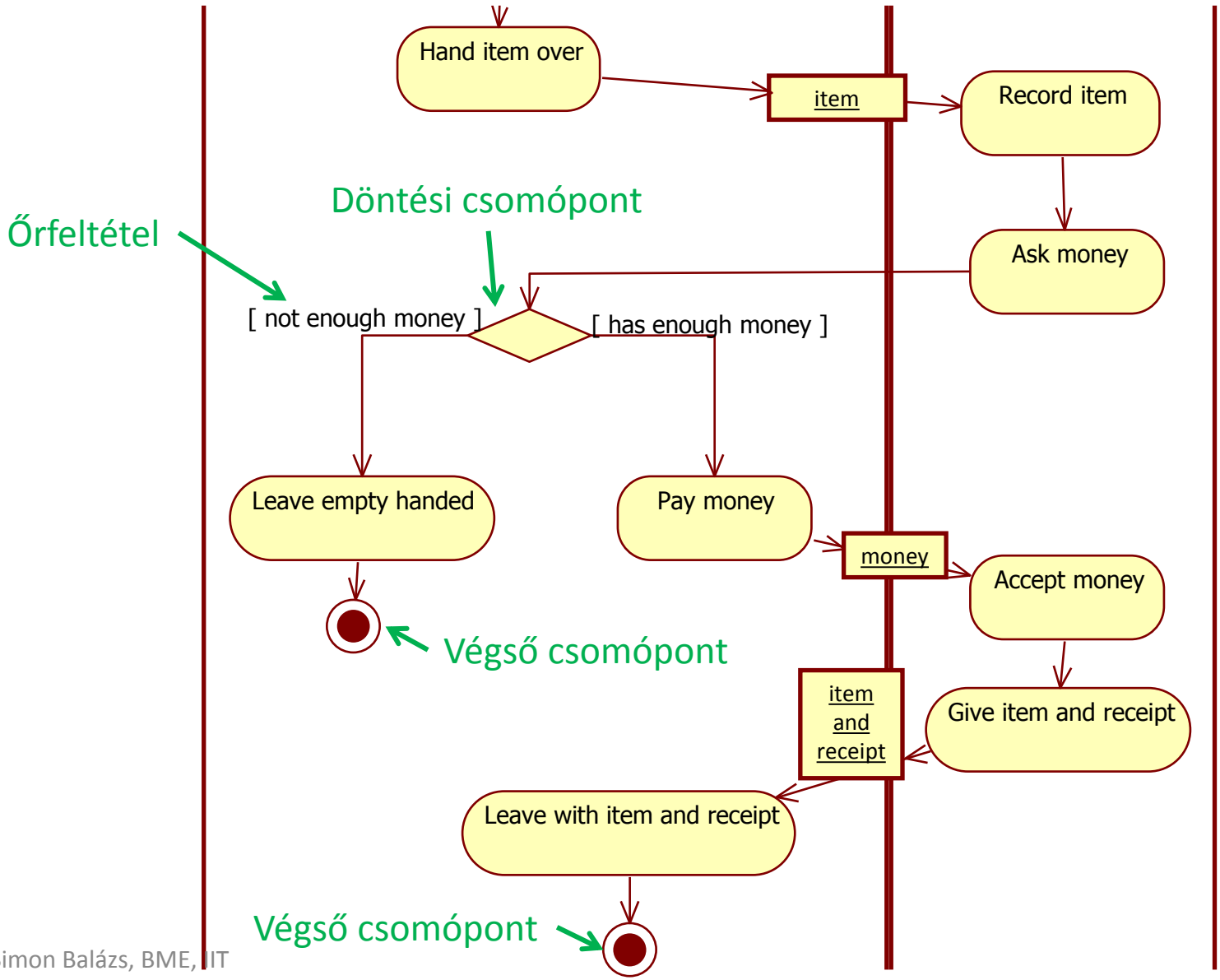
Use case leírás példa: Vásárlás

- **Use case:** Árucikk vásárlása
- **Aktorok:** Vásárló, Pénztáros
- **Főforgatókönyv:**
  - 1. A Vásárló kiválasztja az árucikket
  - 2. A Vásárló átadja az árucikket a Pénztárosnak
  - 3. A Pénztáros lehúzza az árucikket a pénztárgépen
  - 4. A Pénztáros elkéri a pénzt a Vásárlótól
  - 5. A Vásárló kifizeti az összeget a Pénztárosnak
  - 6. A Pénztáros visszaadja az árucikket és a blokkot a Vásárlónak
- **Alternatív forgatókönyv 5.A:**
  - 5.A.1. A Vásárlónak nincs elég pénze
  - 5.A.2. A Vásárló üres kézzel távozik

Aktivitásdiagram példa: Vásárlás use case I.



Aktivitásdiagram példa: Vásárlás use case II.



Aktivitásdiagram elemei

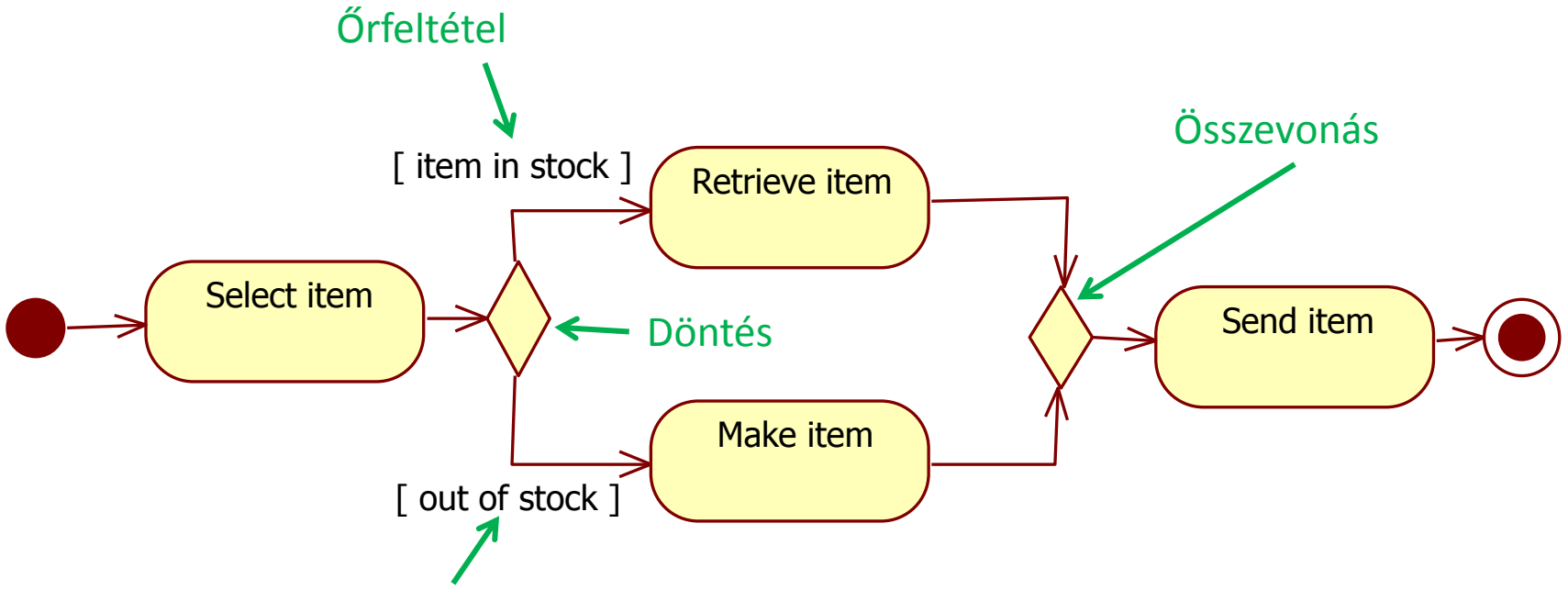
- **Aktivitás (activity)**:
  - az aktivitásdiagram által leírt viselkedés egyes lépéseit modellezi
- **Kezdő csomópont (initial node)**:
  - egy aktivitás futtatásának kezdeteként szolgál
  - nincsenek bejövő élei
  - egy aktivitásnak több kezdő csomópontja is lehet: ilyenkor több futás indul párhuzamosan
- **Vezérlési él (control flow)**:
  - irányított kapcsolat két aktivitás között (forrás és cél)
  - a futás folyamatát jelöli: a cél aktivitás akkor indul, amikor a forrás aktivitás befejeződött
- **Végső csomópont (final node)**:
  - a végső csomópontban a futás befejeződik
- **Döntés (decision)**:
  - a kimenő vezérlési élek közül választ az őrfeltételeik alapján
  - legfeljebb egy kimenő él kerül végrehajtásra
- **Őrfeltétel (guard condition)**:
  - egy vezérlési élhez tartozó feltétel
  - a vezérlési él csak akkor válhat aktívvá, ha az őrfeltétel igaz

Aktivitásdiagram elemei

- **Partíció (partition)**:
  - egy felhasználó (szerepkör) vagy a rendszer egy része egy adott dimenzió mentén
  - jelölése: egy sáv
  - a sávban lévő aktivitásokat az adott felhasználó (szerepkör) vagy részrendszer hajtja végre
  - a partíciók hierarchikusak is lehetnek: alpartíciók
- **Partíció neve (partition name)**:
  - az adott partíció által reprezentált felhasználó (szerepkör) vagy részrendszer neve
  - a sáv fejlécére van írva
- **Objektum (object)**:
  - aktivitások között átadható adatot reprezentál
- **Objektum él (object flow)**:
  - adatokat visz át aktivitások között

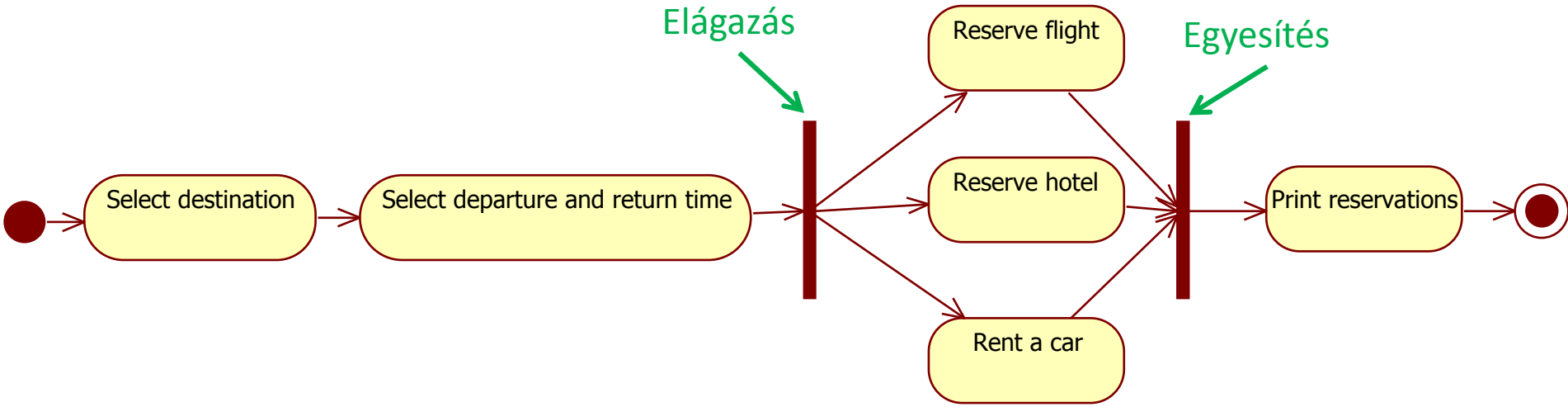
Aktivitásdiagram: döntés és összevonás

- **Döntés (decision)**:
  - a kimenő vezérlési élek közül választ az őrfeltételeik alapján
  - legfeljebb egy kimenő él kerül végrehajtásra
- **Összevonás (merge)**:
  - több vezérlési ág összevonása szinkronizáció nélkül
  - ha több beérkező ág is aktív, az összevonás aktivitás többször is lefut



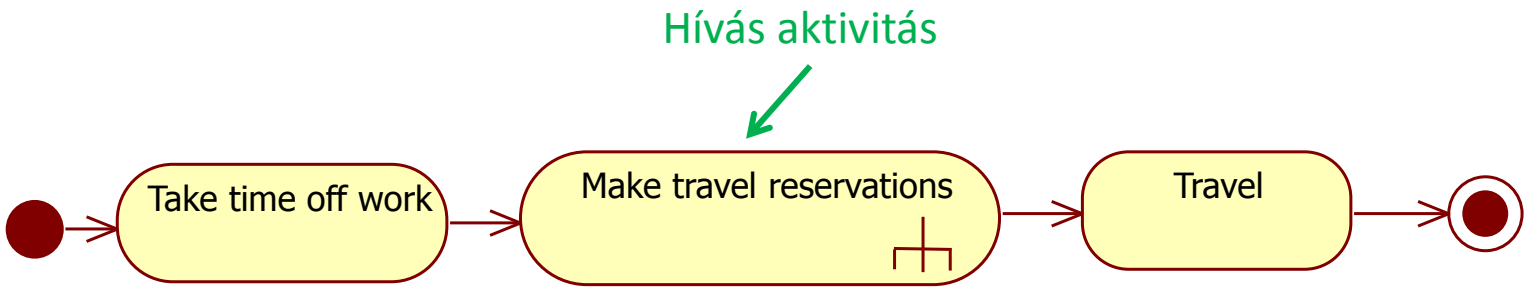
Aktivitásdiagram: elágazás és egyesítés

- **Elágazás (fork)**:
  - a vezérlés több párhuzamosan futó ágra bomlik
- **Egyesítés (join)**:
  - szinkronizálja (bevárja) az összes beérkező ágat
  - alapértelmezetten az összes beérkező ágnak be kell fejeződnie, hogy a vezérlés továbbléphessen
    (ez a viselkedés felülbírálható egy egyéni egyesítő kifejezéssel)



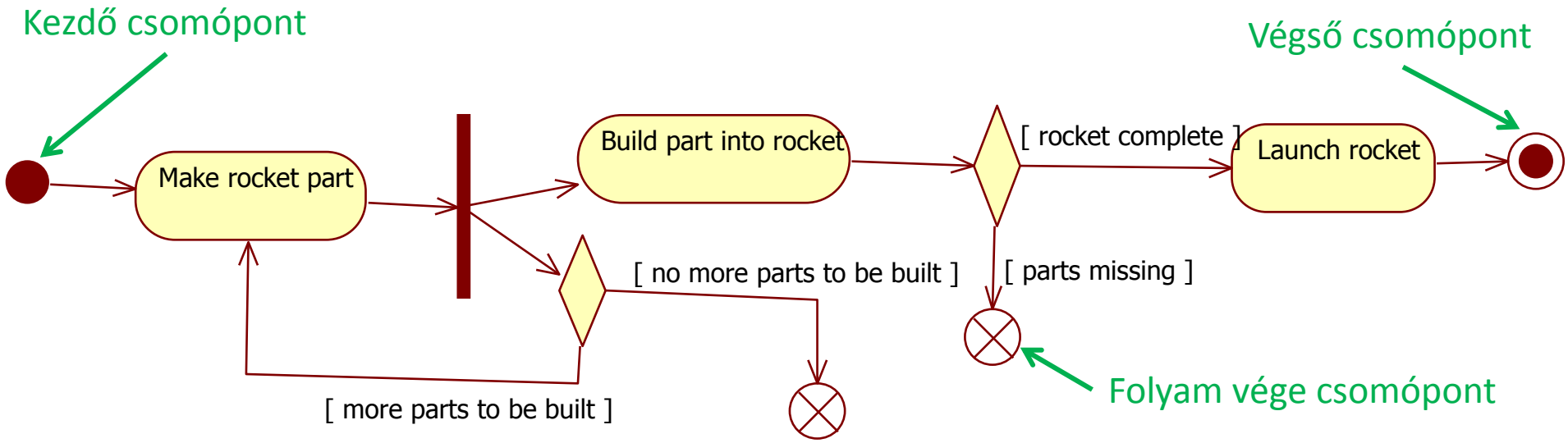
Aktivitásdiagram: hívás aktivitás

- **Hívás aktivitás (call activity)**:
  - a hívás aktivitás meghív egy másik aktivitást
  - a meghívott aktivitás viselkedését egy másik aktivitásdiagram is leírhatja
  - azaz: aktivitásdiagramok egymásba ágyazhatók
  - az aktivitásdiagramok így különböző szintű részletezettséget mutathatnak



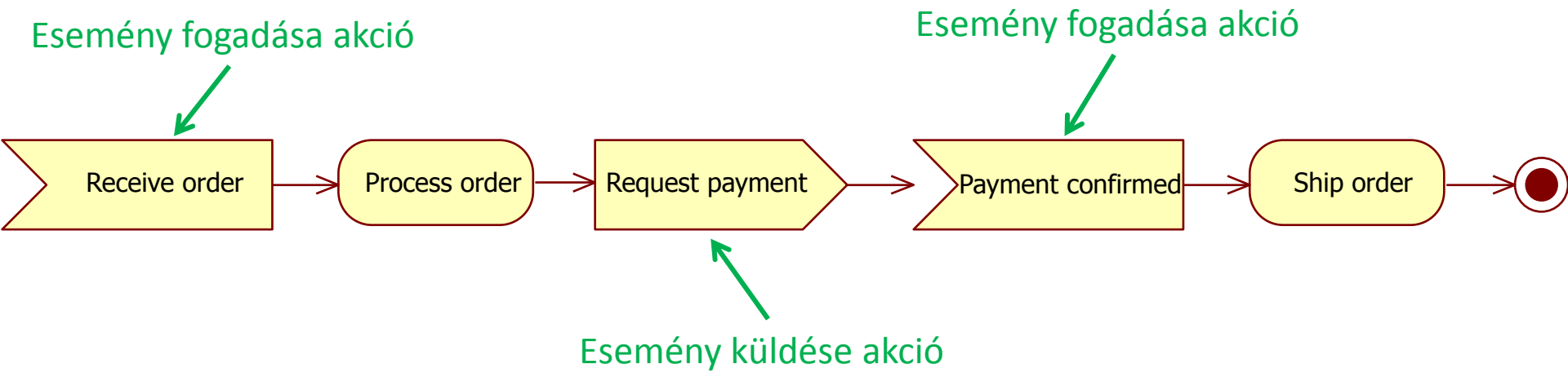
Aktivitásdiagram: végső csomópont

- Kétfajta végső csomópont létezik:
  - **Végső csomópont aktivitás (activity final node)**: befejezi az aktivitásdiagram által leírt viselkedést, minden vezérlési ág befejeződik
  - **Folyam vége csomópont (flow final node)**: csak az adott vezérlési ág fejeződik be, a többi vezérlési ág fut tovább



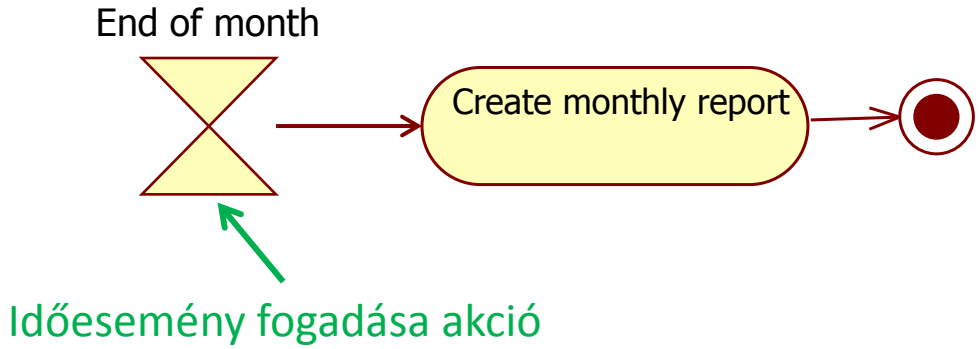
Aktivitásdiagram: jelzések

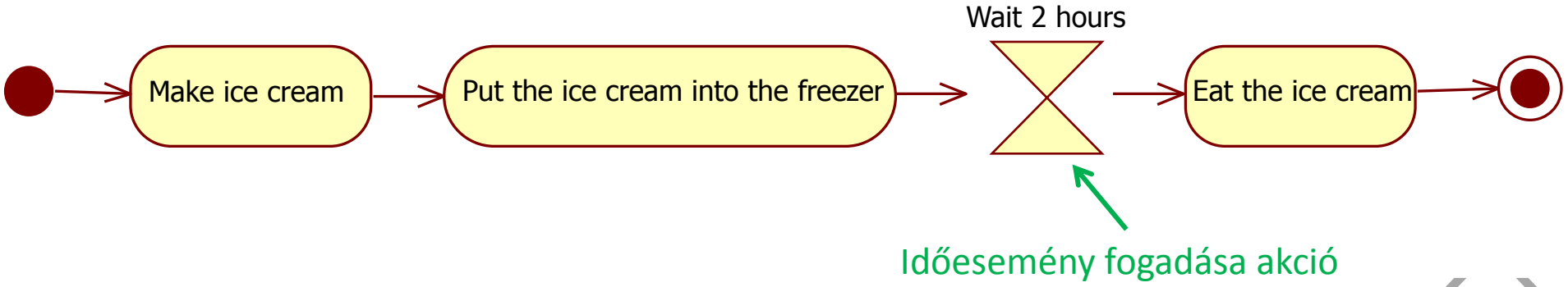
- Aktivitásokat események (jelzések) is elindíthatnak, nem csak a kezdeti csomópont
- Eseményeket az aktivitásdiagram elején és közepén is lehet fogadni
- A jelzések kezelésével kapcsolatos akciók:
  - **Esemény fogadása akció (accept event action)**: megvárja, amíg az esemény (jelzés) bekövetkezik, majd elindítja a vezérlés futását
  - **Esemény küldése akció (send signal action)**: jelzést küld, amely egy esemény fogadása akcióval elkapható



Aktivitásdiagram: időzítési események

- **Időesemény fogadása akció (accept time event action)**: elindítja a vezérlést egy adott időpontban
- Példák:





UML diagramok típusai

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

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

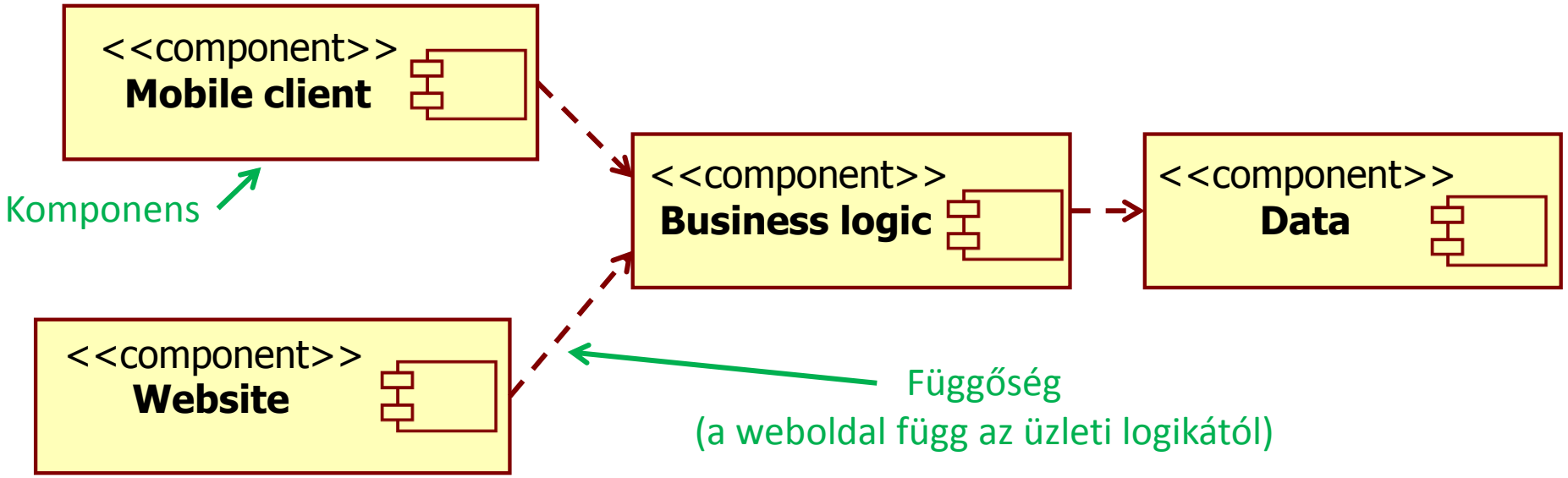
Komponensdiagram (Component Diagram)

Komponensdiagram

- Egy szoftverkomponens a kompozíció alapegysége, szerződésszerűen specifikált interfészekkel rendelkezik és csak explicit módon függ környezetétől. Egy szoftverkomponens önállóan telepíthető és harmadik fél által kompozíciós egységként felhasználható. (Szyperski)
- A komponensdiagram egy rendszer komponenseit és a közöttük lévő kapcsolatokat mutatja
- A komponensdiagram tipikusan tervezés közben készül, de finomítható telepítés és futtatás során

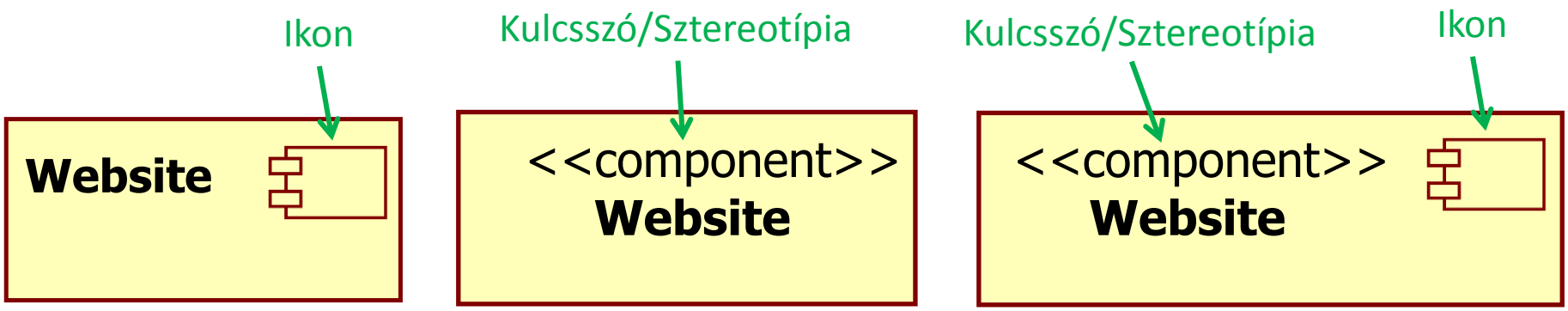
Komponensdiagram példa: Mobil+webes alkalmazás

- Az áttekintő komponensdiagram a komponenseket (component) és a közöttük lévő függőségeket mutatja
- **Függőség (dependency)**: “A” függ “B”-től azt jelenti, hogy ha “B” változik, akkor a változás kihathat “A”-ra is
- A függőség fajtája nincs jelölve
- A függőséget jelentő nyíl hiánya egyben a függőség hiányát is jelzi



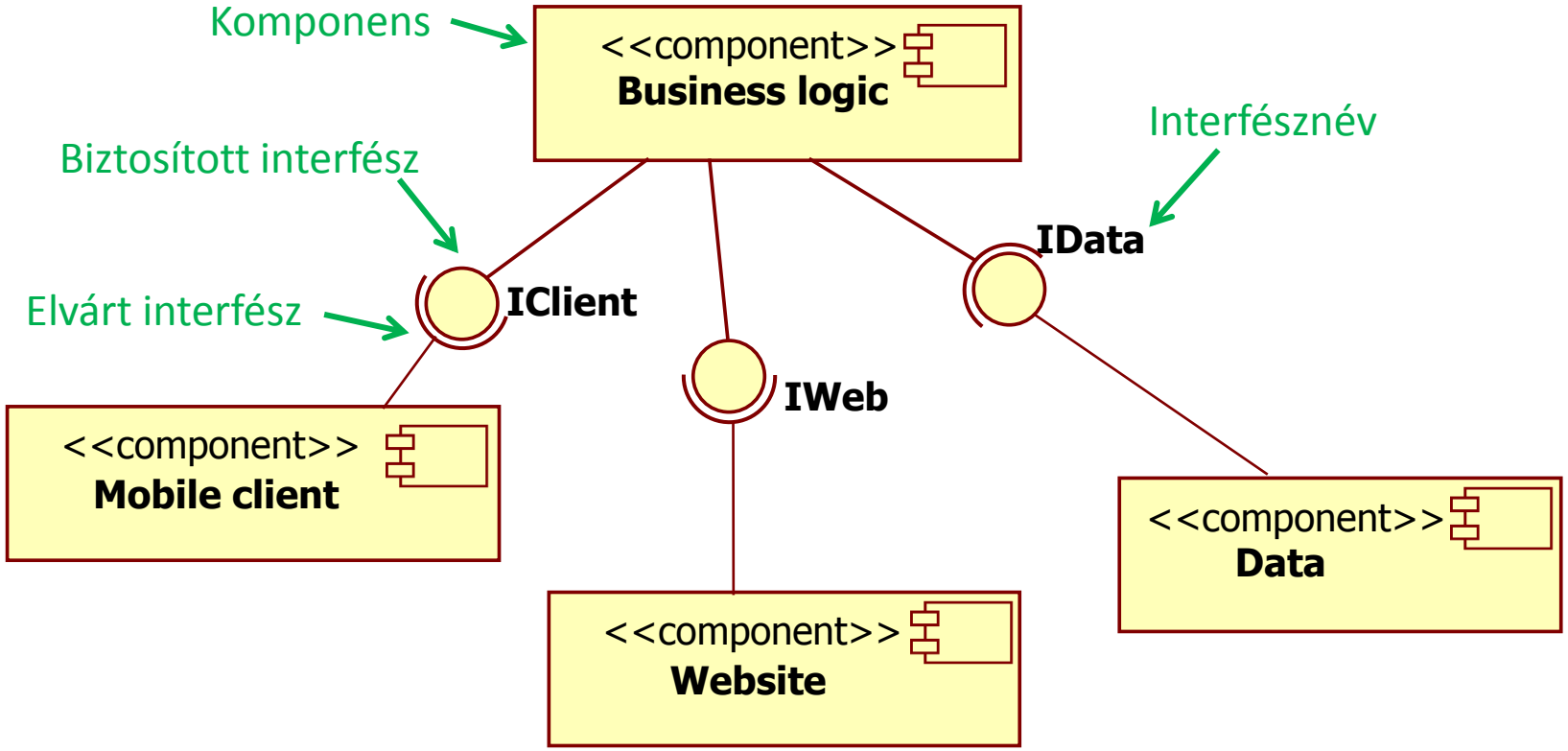
Komponens jelölése

- Három lehetséges jelölési mód (ekvivalensek egymással):
  - téglalap ikonnal
  - téglalap *component* kulcsszóval/sztereotípiával (keyword/stereotype)
  - téglalap ikonnal és *component* kulcsszóval/sztereotípiával



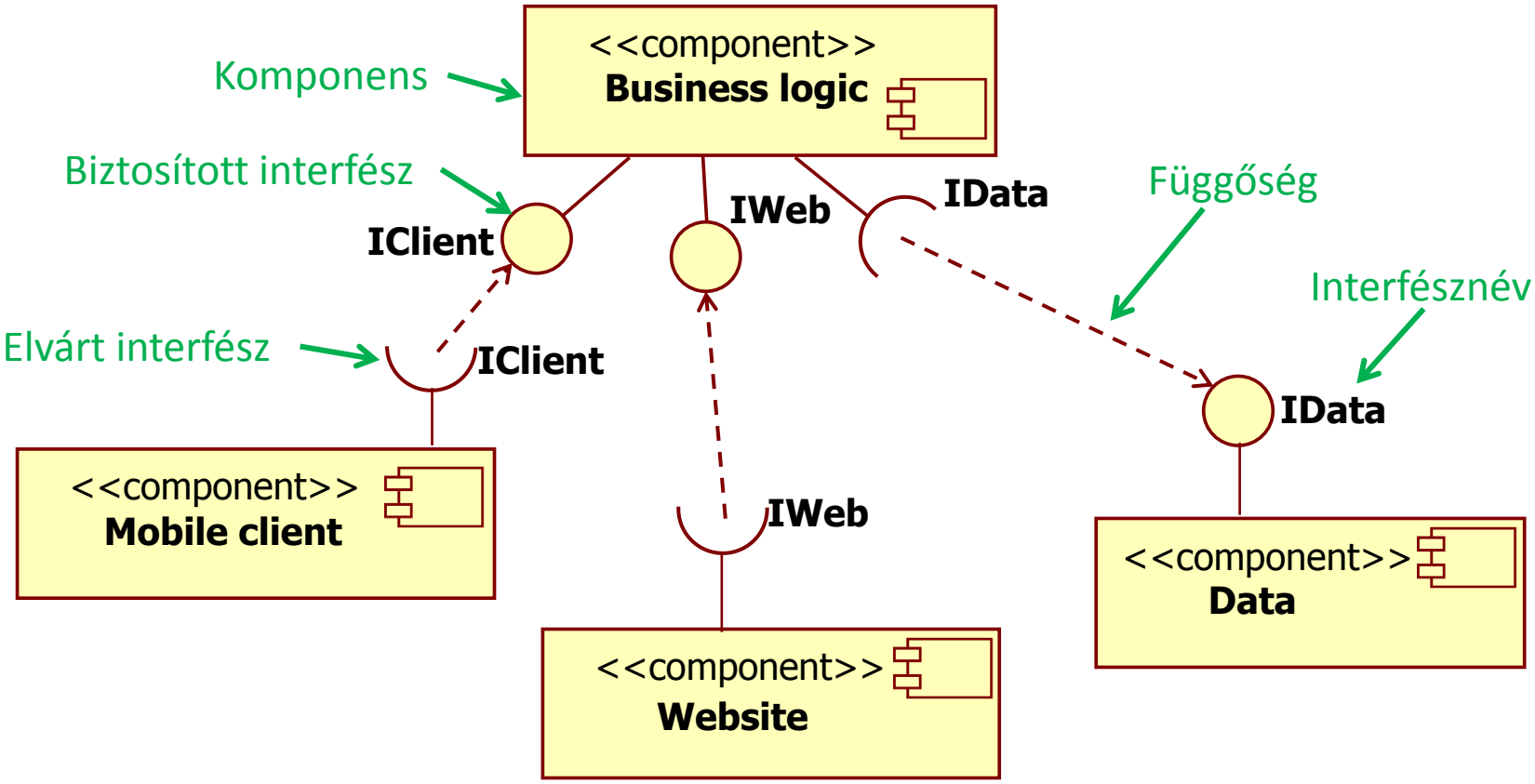
Komponensdiagram példa: Mobil+webes alkalmazás

- Egy részletesebb komponensdiagramon ábrázolhatók a komponensek a biztosított (provided) és elvárt (expected) interfészeikkel, valamint a közöttük lévő kapcsolatokkal



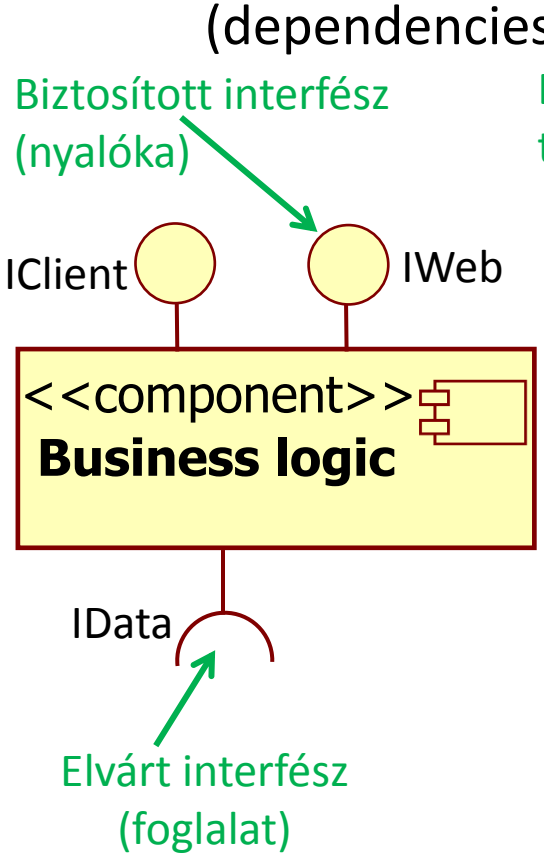
Komponensdiagram példa: Mobil+webes alkalmazás

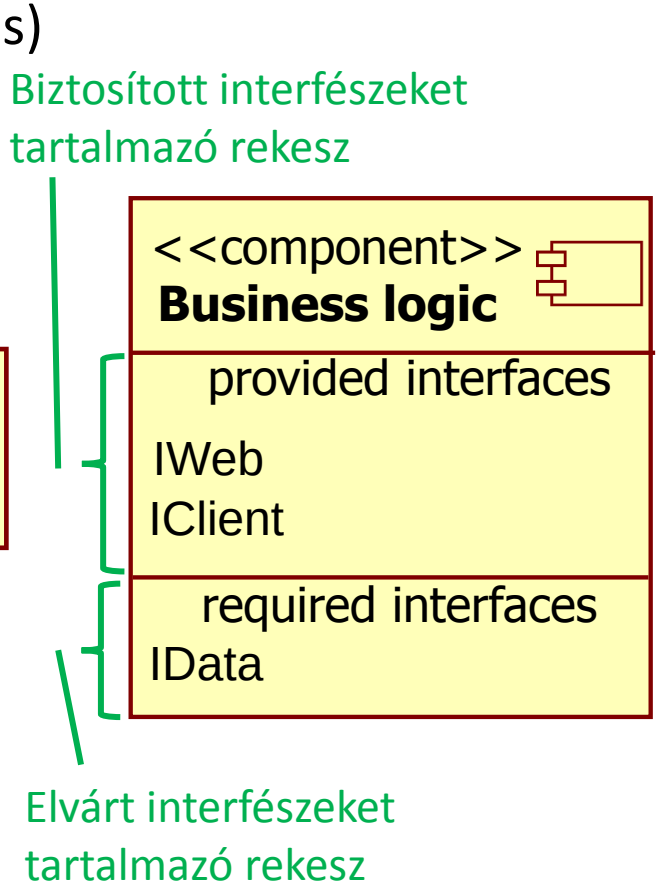
- A biztosított és elvárt interfészek függőség (dependency) segítségével is összekapcsolhatók

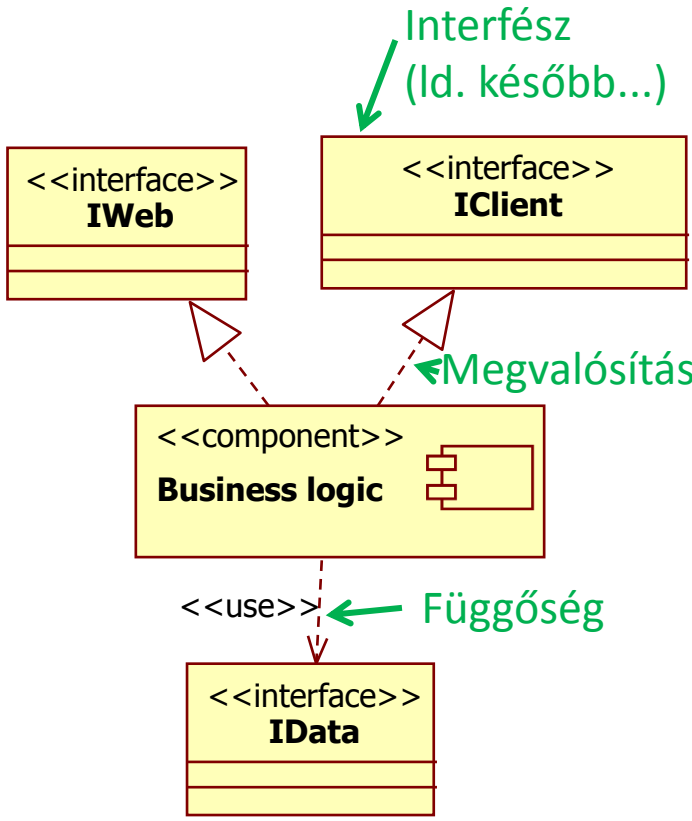


Biztosított és elvárt interfészek

- Három lehetséges jelölés (ekvivalensek egymással):
  - komponens nyalókákkal (lollipops) és foglalatokkal (sockets)
  - komponens rekeszekkel (compartments)
  - komponens megvalósításokkal (realizations) és függőségekkel (dependencies)

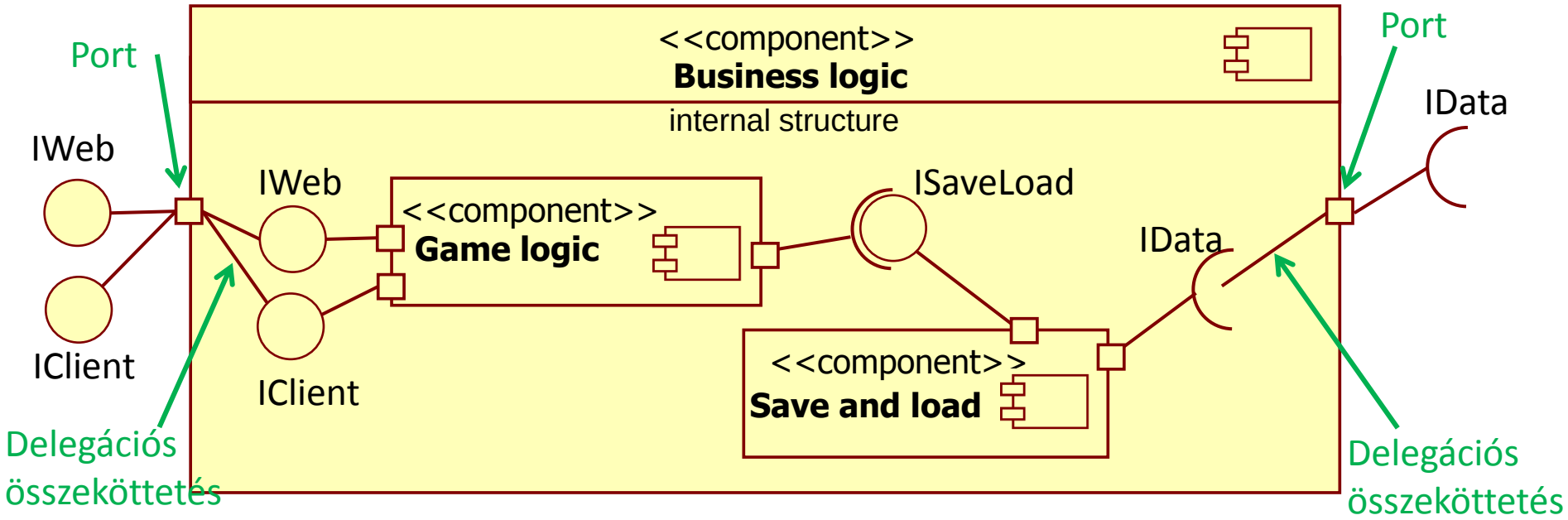






Összetett komponens (composite component)

- Az összetett komponenst más komponensek implementálják
- A külsőleg biztosított és elvárt interfészek portokon keresztül vannak kivezetve
- Delegációs összeköttetések (delegation connectors) kapcsolják össze a külsőleg biztosított és elvárt interfészeket és a belső komponenseket, amelyek megvalósítják vagy elvárják őket



UML diagramok típusai

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

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

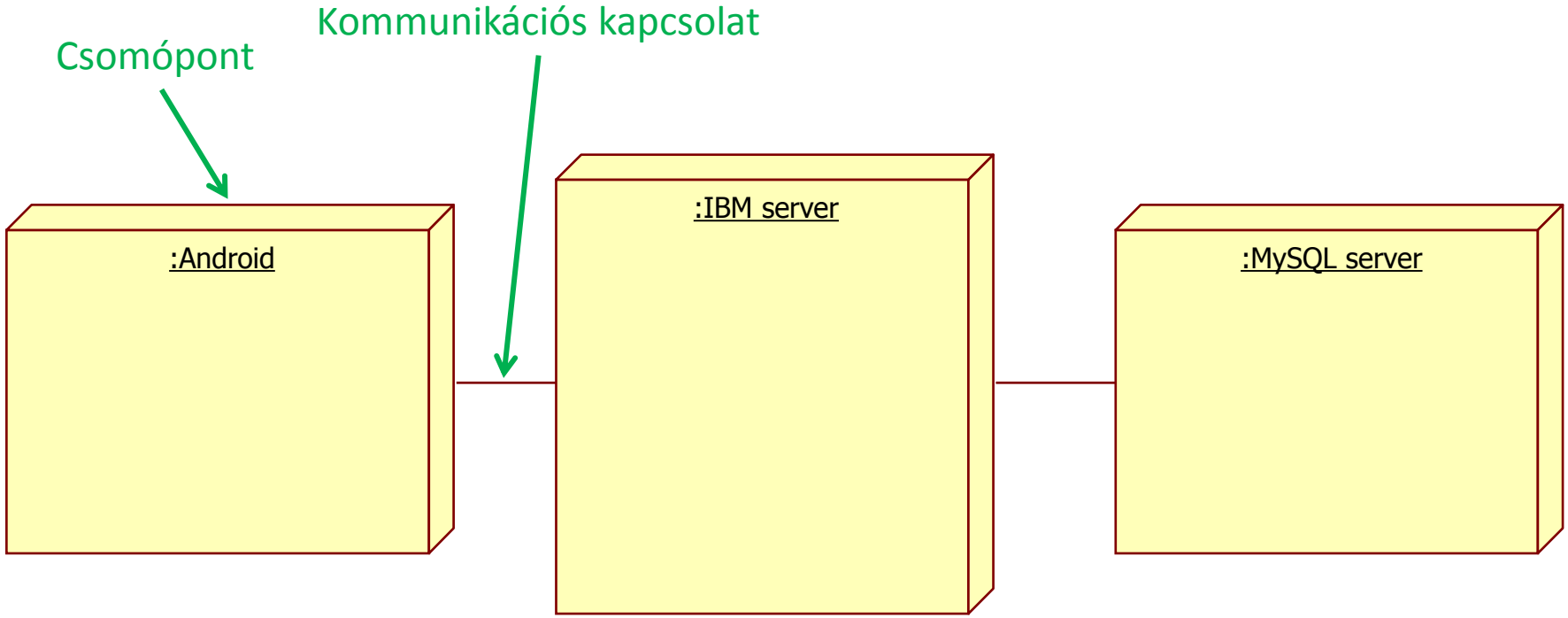
Telepítési diagram (Deployment Diagram)

Telepítési diagram

- A telepítési diagram a rendszer futási architektúráját és szoftver artifact-ek rendszerelemekhez való rendelését ábrázolja
- A telepítési diagram segítségével a rendszer hardveres és szoftveres topológiája is ábrázolható
- A telepítési diagramot tipikusan átadáskor használjuk a telepítési architektúra megtervezésére

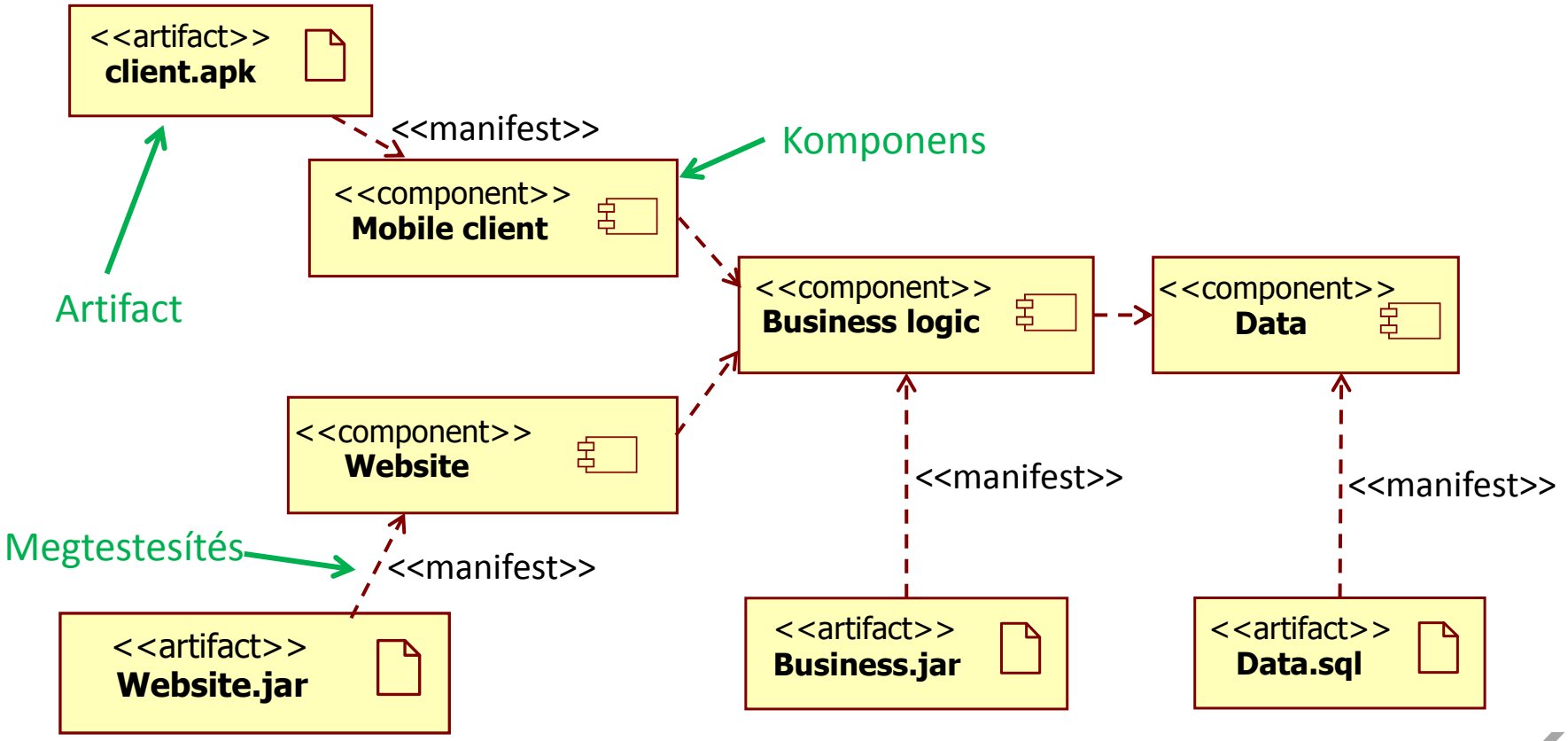
Telepítési példa: Mobil+webes alkalmazás

- A rendszer architektúráját csomópontokkal (nodes) és a közöttük lévő kommunikációs kapcsolatokkal (communication path) írhatjuk le



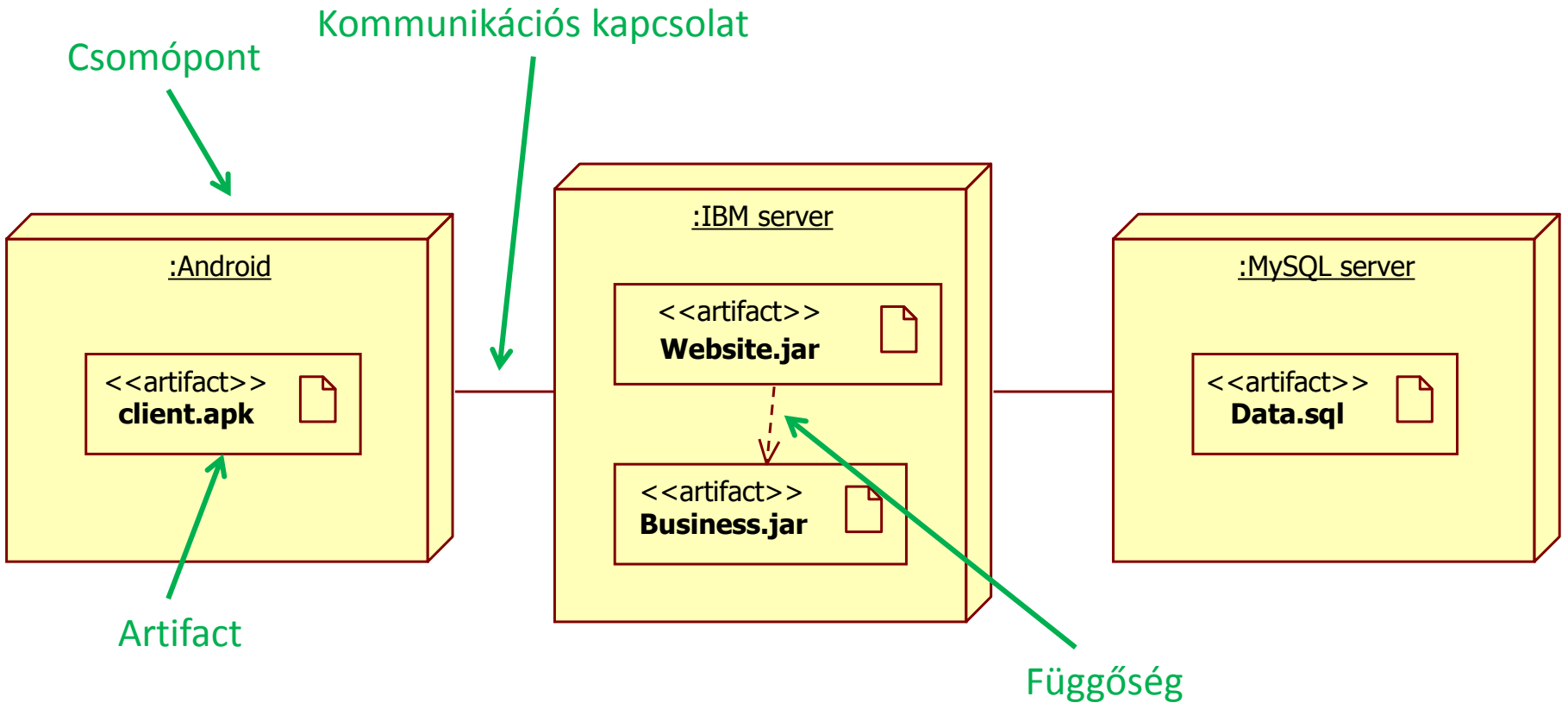
Artifact példa: Mobil+webes alkalmazás

- Egy artifact olyan dolgot reprezentál, amelyet vagy a szoftverfejlesztési folyamat vagy pedig a rendszer működése állít elő vagy használ fel
  - pl. futtatható fájl, szkript, adatbázis tábla, dokumentum
- Egy artifact egy komponens megtestesítését is jelentheti



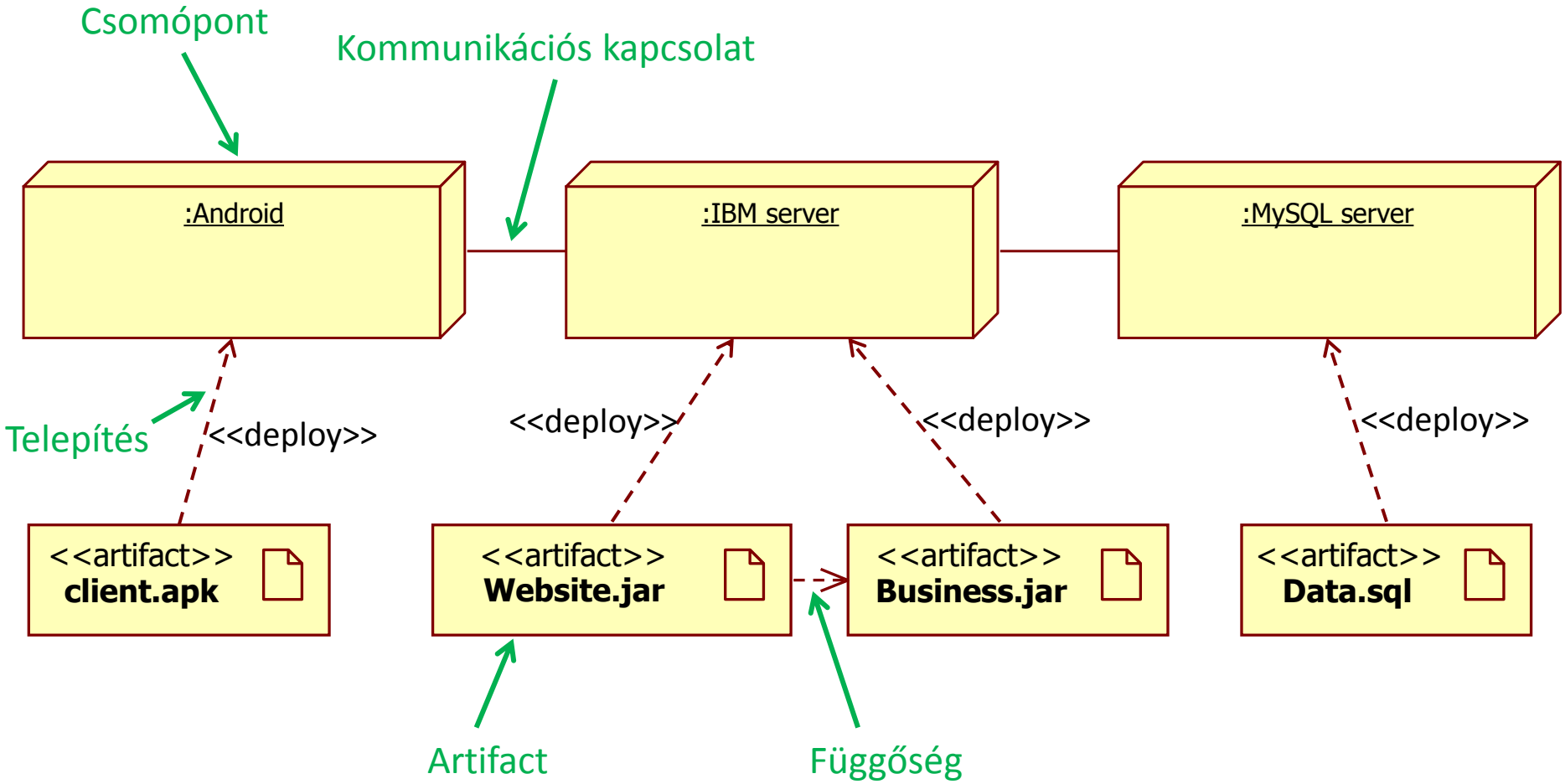
Telepítési példa: Mobil+webes alkalmazás

- Artifact-eket csomópontokra lehet telepíteni
- Függőségek artifact-ek között is lehetnek



Telepítési példa: Mobil+webes alkalmazás

- A telepítendő artifact-okat deploy kulcsszóval/sztereotípiával ellátott függőség segítségével is lehet csomóponthoz rendelni
  - (ez a diagram ekvivalens az előző dián szereplővel)



UML diagramok típusai

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

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

Köszönöm a figyelmet!

Unified Modeling Language

Szoftvertechnológia

Dr. Goldschmidt Balázs

BME, IIT

Tartalom

- UML diagramok:
  - 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:
Hogyan tervezzük meg egy komponens belsejét?
(Struktúra és viselkedés)

Hol tartunk?

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

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

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

Osztálydiagram (Class Diagram)

Osztálydiagram

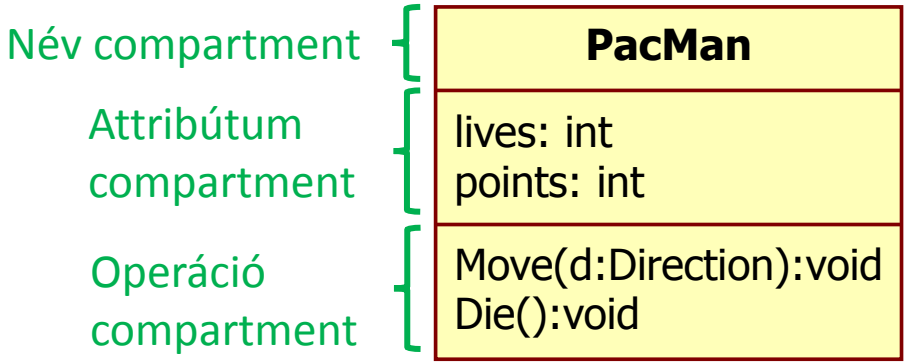
- Objektumorientált modellek leírásának szabványos módja
- A leggyakrabban használt UML diagram
- Tipikusan szoftverfejlesztők használják a szoftver architektúrájának dokumentálására
- Az osztálydiagramok közvetlenül programkóddá alakíthatók
- Vigyázat:
  - Az osztálydiagram független a programozási 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))



C++/Java/C# leképzés:

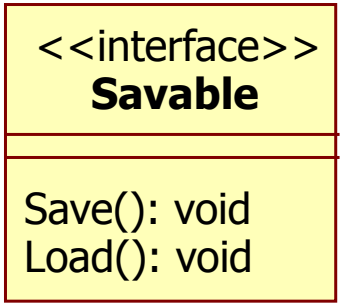
```
class PacMan {
  int lives;
  int points;

  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++ leképzés:

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

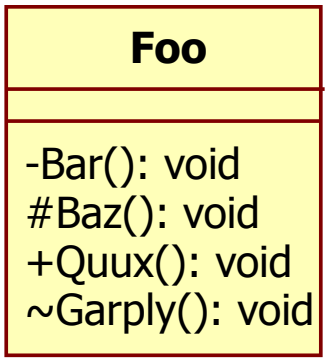
Java/C# leképzés:

```
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



C++ leképzés:

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

Java leképzés:

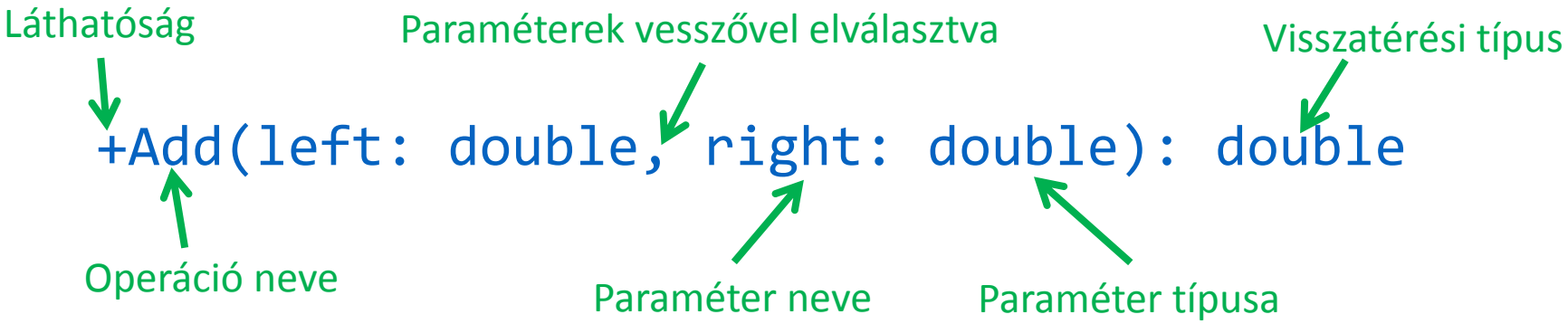
```
public class Foo {
  private void Bar() { /* ... */ }
  protected void Baz() { /* ... */ }
  public void Quux() { /* ... */ }
  void Garply() { /* ... */ }
}
```

C# leképzés:

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

Operációk (operations)

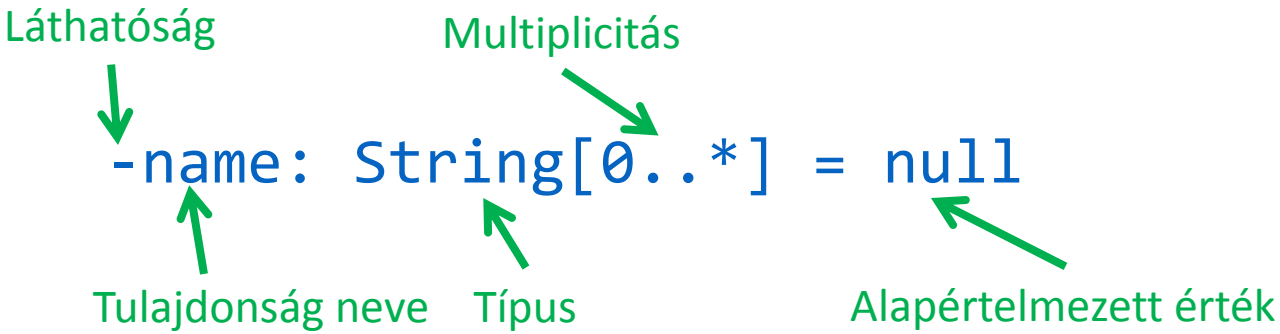
- Az osztály példányainak viselkedését írják le: az objektumok által biztosított szolgáltatásokat
- Egy operáció szignatúrája:



- 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:



- 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

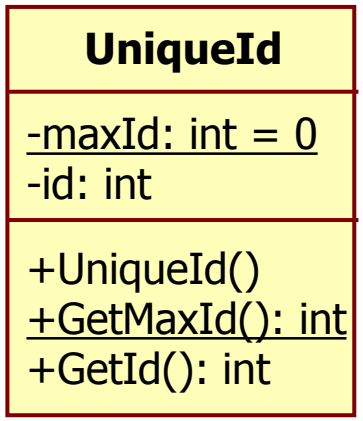
- Tagfüggvények (member operation) és -tulajdonságok (member property) az osztály példányaihoz kapcsolódnak
  - példány-szintűnek (instance-scope) is hívják őket
- A tagtulajdonságok értékei példányonként különböznek
  - ha az egyik példányban megváltozik az értékük, az nincs kihatással a többi példányra
- 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 elérhetők 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:



Java/C# leképzés:

```
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;
  }
}
```

C++ leképzés:

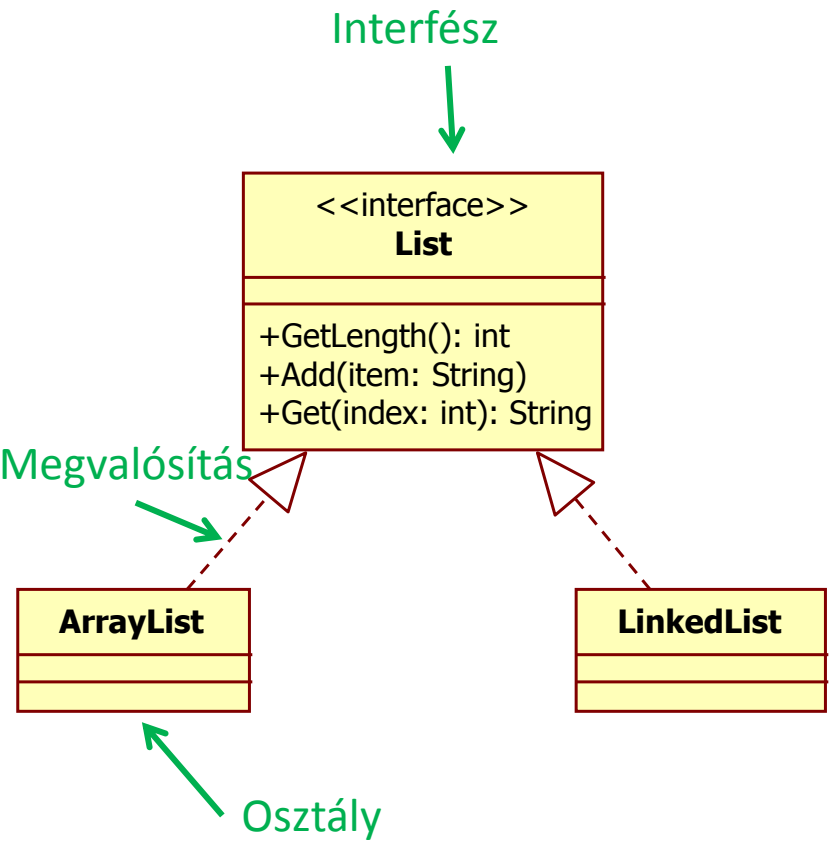
```
class UniqueId {
private:
  static int maxId;
  int id;
public:
  UniqueId() {
    this->id = ++UniqueId::maxId;
  }
  static int GetMaxId() {
    return UniqueId::maxId;
  }
  int GetId() {
    return this->id;
  }
};

int UniqueId::maxId = 0;
```

(A függvények törzsét nekünk kell megírni, ezek nem olvashatók ki az osztálydiagramból.)

Megvalósítás (realization)

- Osztályok implementálhatnak/megvalósíthatnak (realize) interfészeket



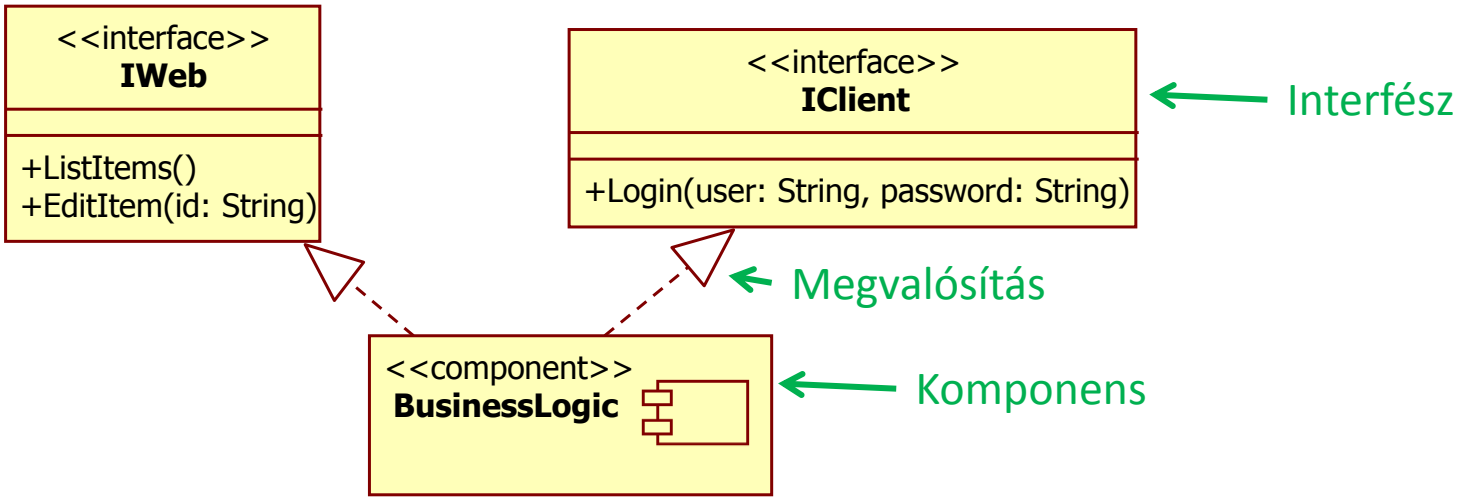
Java leképzés:

Megvalósítás

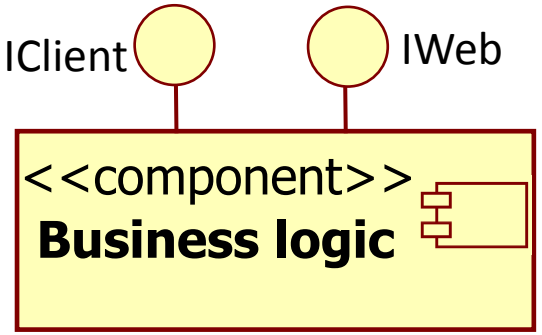
```
public class ArrayList implements List
{
    public int GetLength()
    {
        // ...
    }
    public void Add(String item)
    {
        // ...
    }
    public String Get(int index)
    {
        // ...
    }
}
```

Megvalósítás (realization)

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



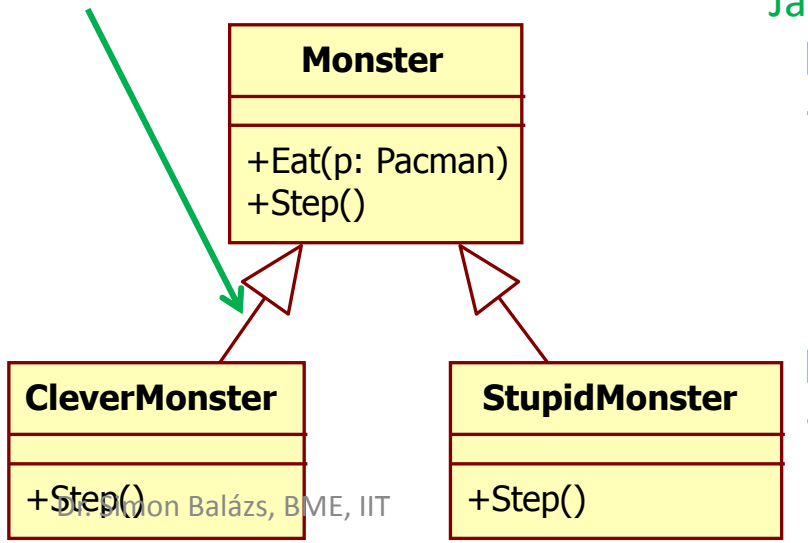
Ekvivalens a nyalóka jelöléssel:
a nyalóka jelölésnél azonban nem mutatjuk az interfész operációit



Öröklődés (generalization)

- Öröklődés 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 felüldefiniálhatják a leszármazottak
  - így lehet kiterjeszteni az ős viselkedését
  - az UML-ben minden tagfüggvény virtuális

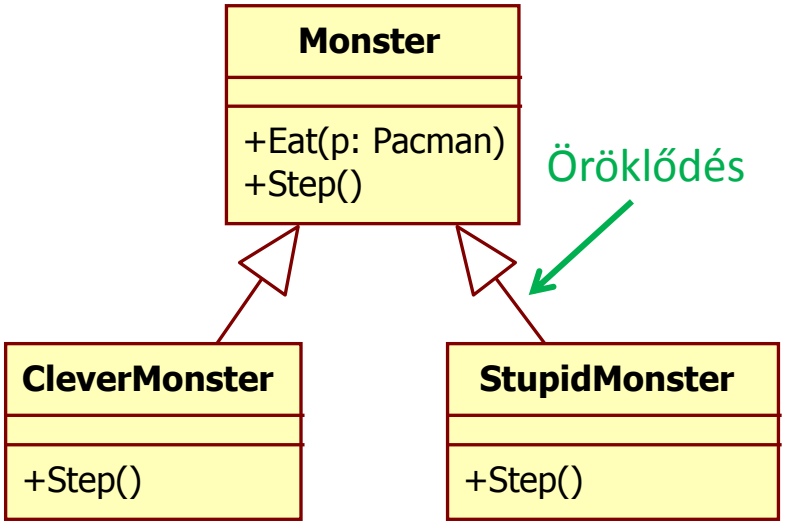
Öröklődés



Java leképzés:

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

Öröklődés (generalization)



C++ leképzés:

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

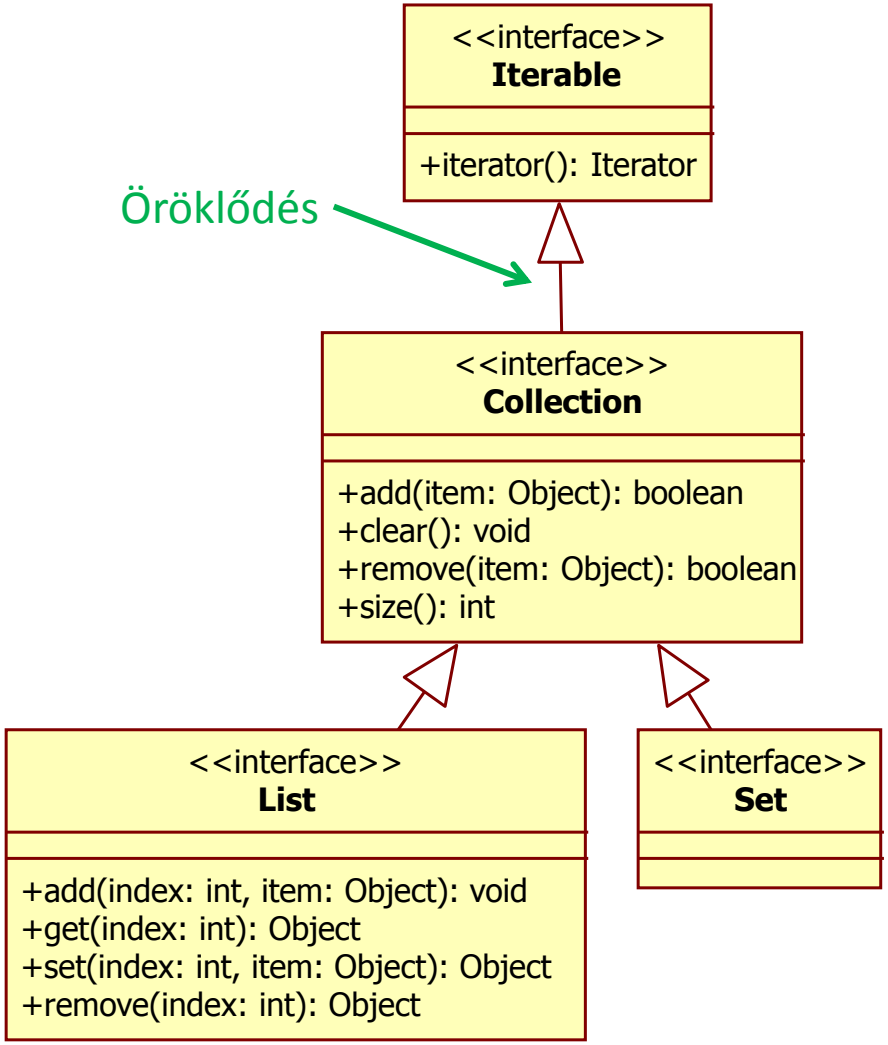
C# leképzés:

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

Öröklődés (generalization)

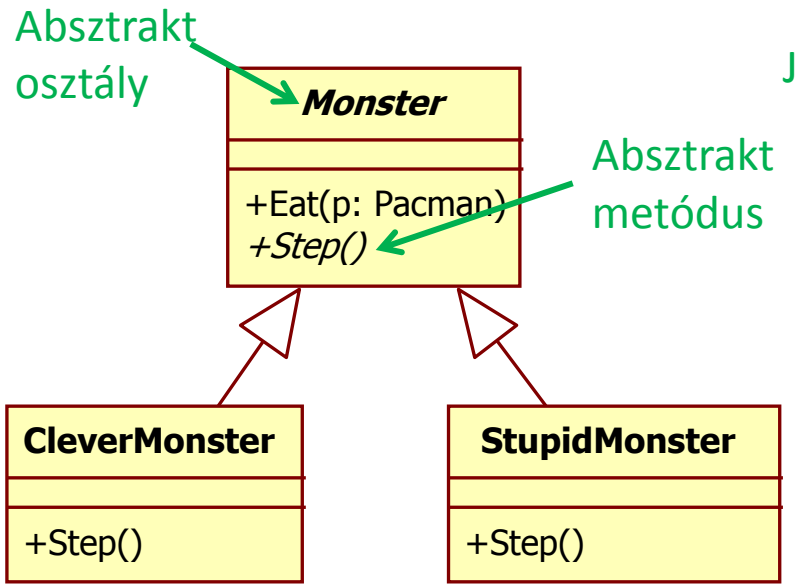
- Öröklődés interfészek között
  - többszörös is megengedett
- Az öröklődés “az-egy” reláció a leszármazott és az ős interfész között
- A leszármazott interfész újrahasznosítja és kibővíti az ős által definiált viselkedést

Példa: Java kollekciók



Absztrakt operációk és absztrakt osztályok

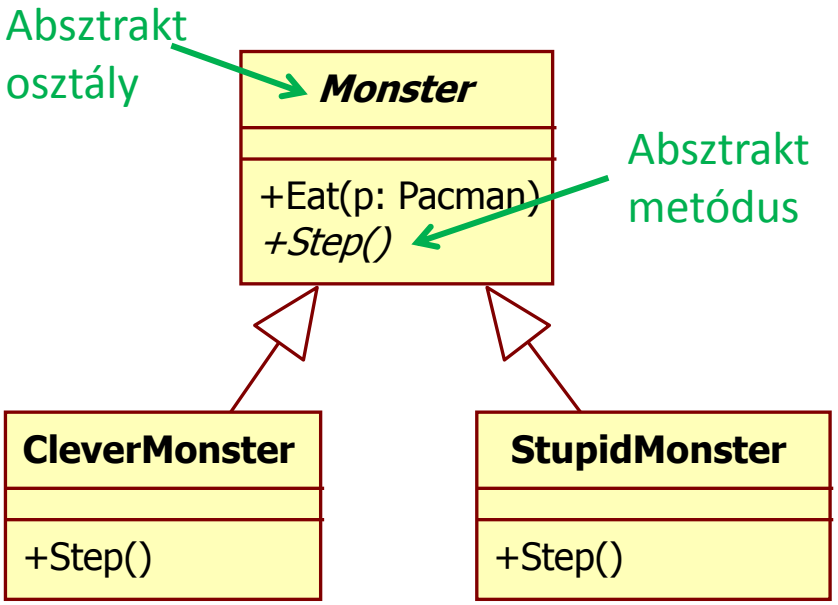
- Absztrakt operáció (abstract operation):
  - virtuális függvény implemenráció nélkül
  - egy konkrét (nem absztrakt) leszármazottnak kell hogy legyen implementációja erre a függvényre
- Absztrakt osztály (abstract class):
  - nem példányosítható
  - általában van legalább egy absztrakt függvénye, de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



Java leképzés:

```
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



C++ leképzés:

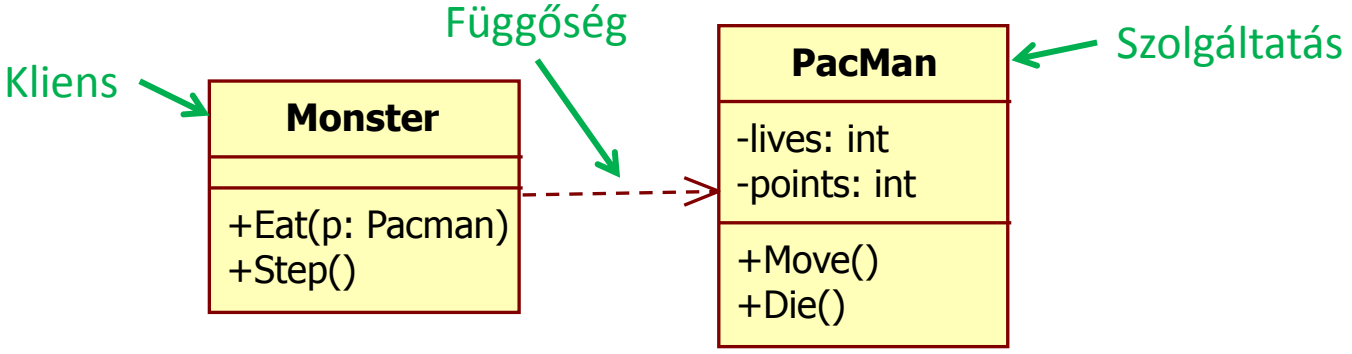
```
class Monster
{
  public void Eat(Pacman* p) { /*...*/
  public virtual void Step() = 0;
}
class CleverMonster : public Monster
{
  public void Step() { /*...*/ }
}
```

C# leképzés:

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

Függőség (dependency)

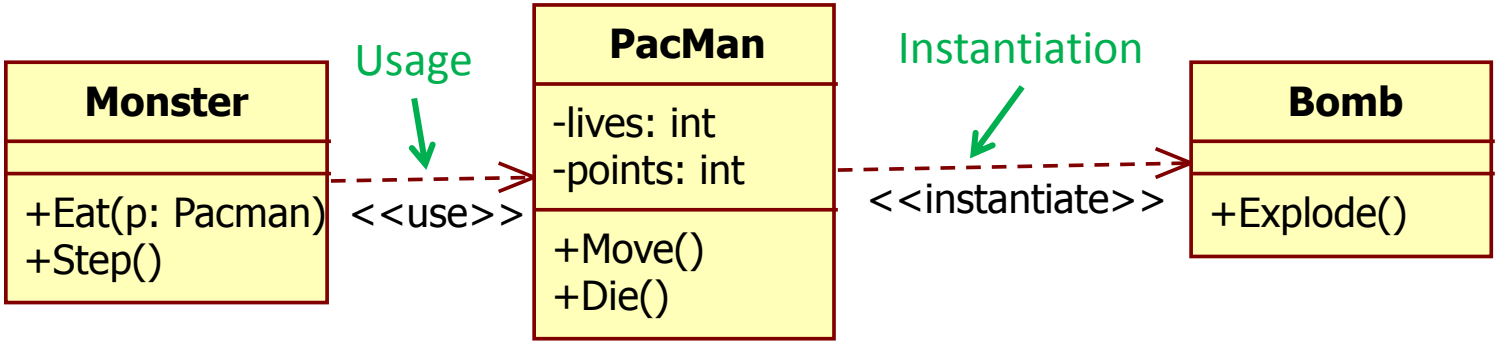
- Egy kliens-szolgáltatás kapcsolatot definiál két modellelem között
- A szolgáltatás megváltozása magával vonhatja a kliens megváltoztatását
- Calssifier-ek között gyenge, ideiglenes kapcsolat:
  - csak egy függvényhívás erejéig tart
  - példák:
    - a kliens paraméterként kap egy példányt a szolgáltatásból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

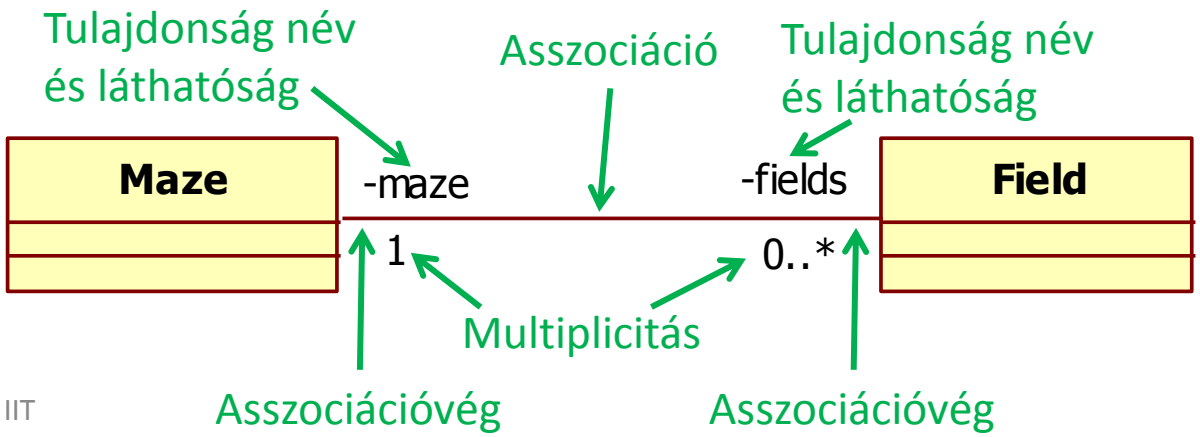
- A függőség jelentése pontosítható sztereotípiával:



  - <<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# leképzés:

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

Java leképzés:

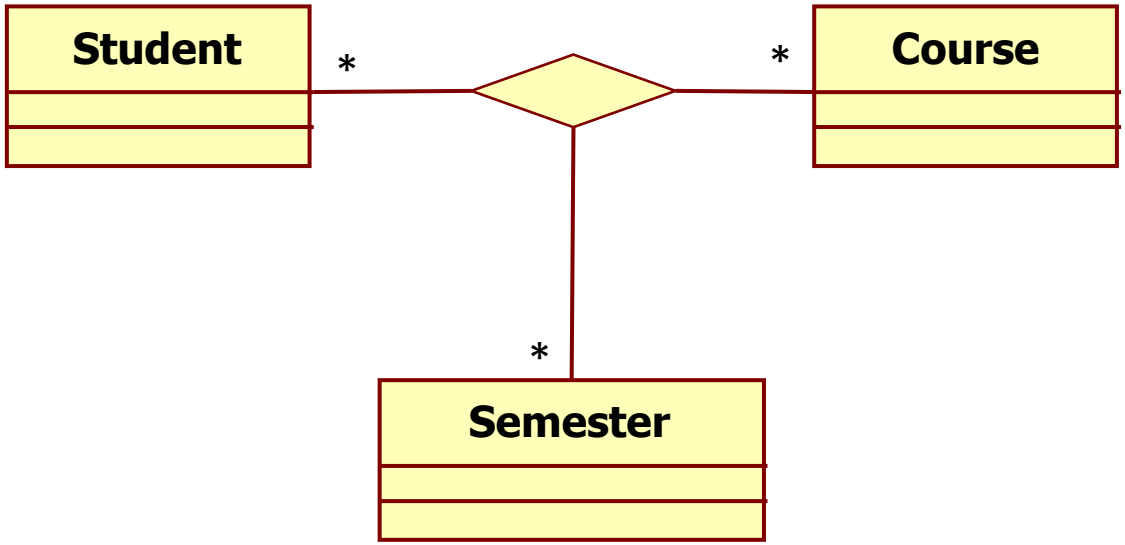
```
public class Maze {
  private ArrayList<Field> fields;
}
public class Field {
  private Maze maze;
}
```

C++ leképzés:

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

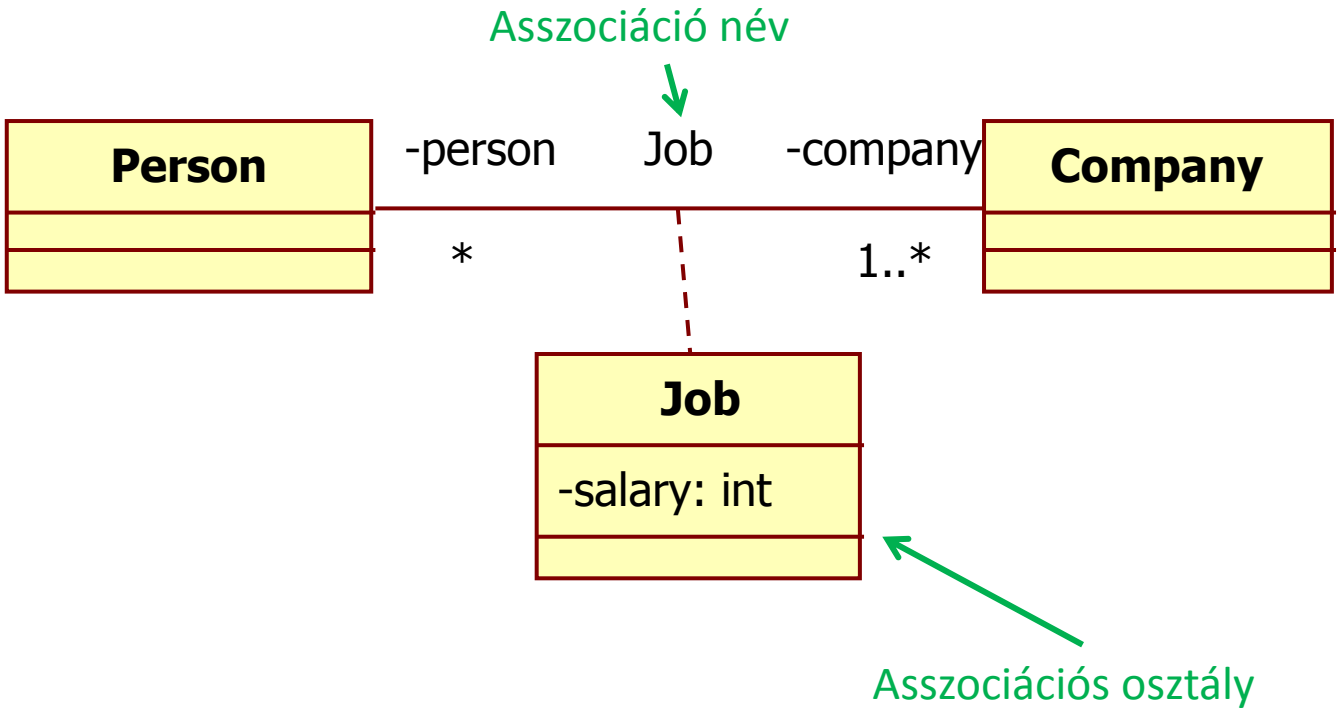
Többvégű asszociáció

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



Asszociációs osztály (association class)

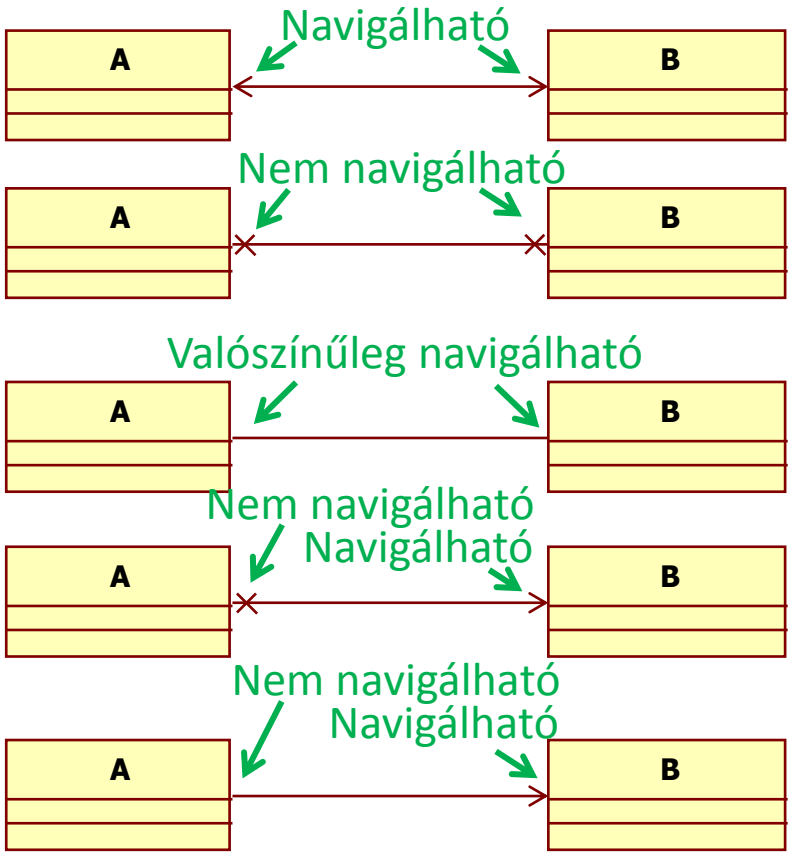
- Egy asszociációnak lehetnek tulajdonságai és operációi
- Egy asszociációs osztály tudja ezeket reprezentálni



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

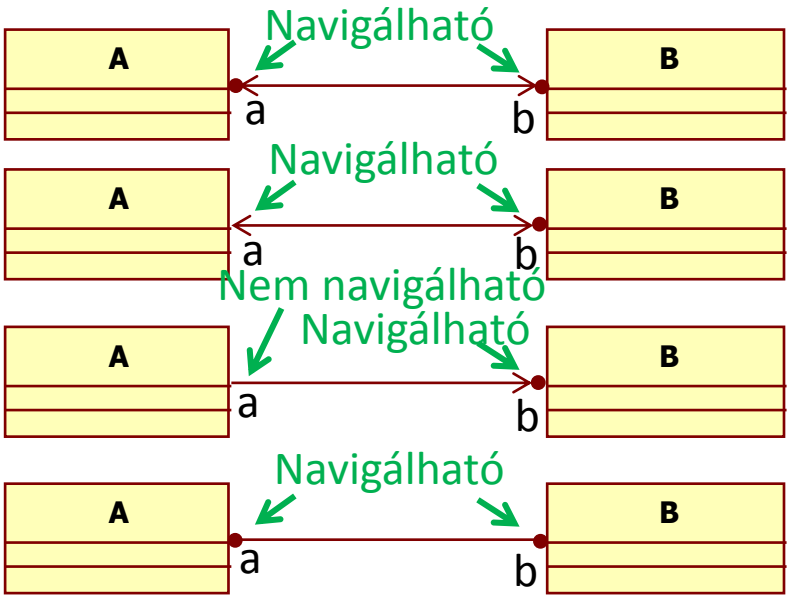
Navigálhatóság (navigability)

- 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



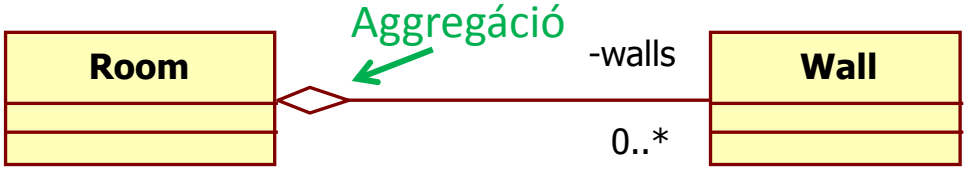
Birtokolt vég (owned end)

- 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



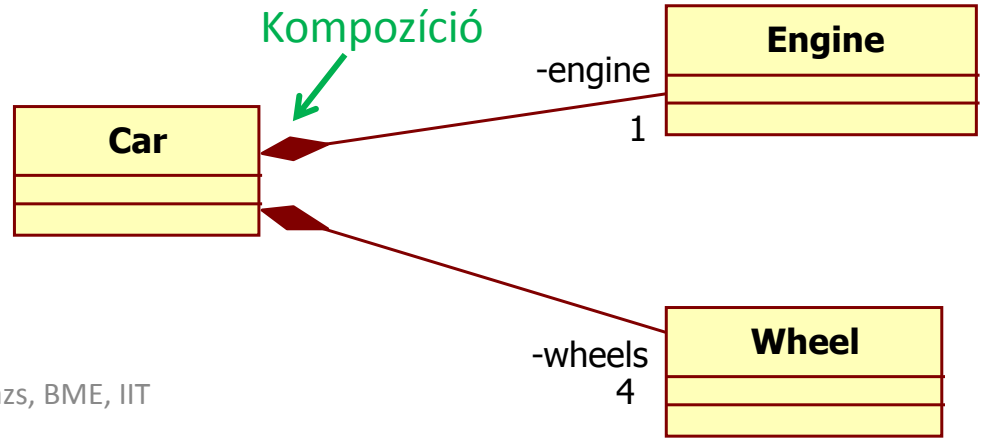
Aggregáció (aggregation), tartalmazás (composition)

- 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

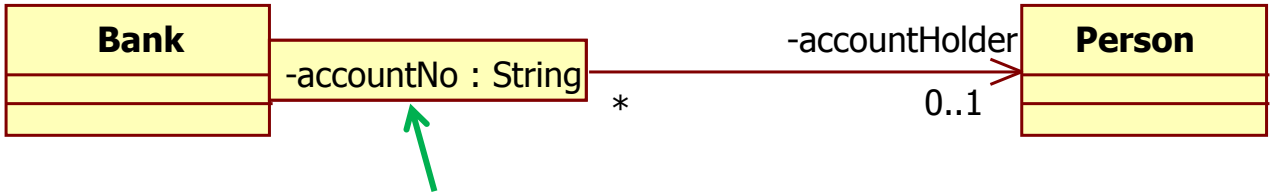


Egy autó tartalmaz egy motort és négy kereket

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!)

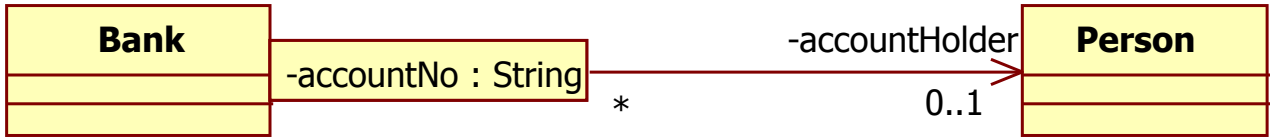


C++ leképzés:

```
class Person
{
}
class Bank
{
private:
  map<string, Person*> accountHolder;
}
```

(Ha több qualifier van, akkor több sorban vannak felsorolva.)

Minősítő (qualifier)



Java leképzés:

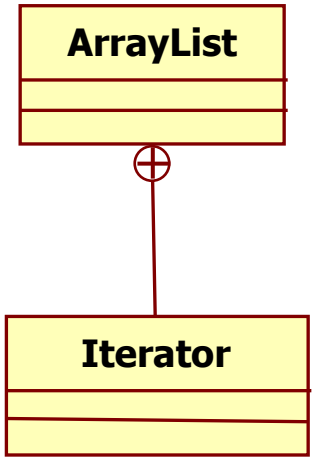
```
public class Person
{
}
public class Bank
{
  private Map<String, Person> accountHolder;
}
```

C# leképzés:

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

Beágyazott osztály (nested class)

- A beágyazott osztályt egy másik osztályon belül definiáljuk



C++ leképzés:

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

C# leképzés:

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

Java leképzés:

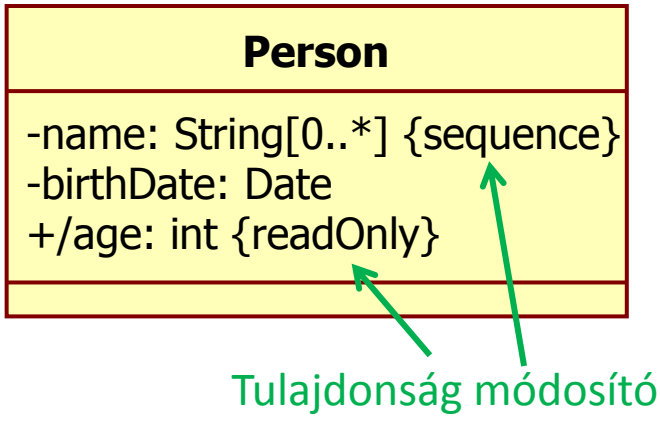
```
public class ArrayList
{
    public static class Iterator
    {
    }
}
```

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# leképzés:

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



Java leképzés:

```
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 módosítók (property modifiers)

Módosító	Jelentés
readOnly	a tulajdonság csak olvasható
union	a tulajdonság értéke a részhalmazaiból számolt unió
subsets <*propname*>	a tulajdonság a <*propname*> részhalmaza
redefines <*propname*>	a tulajdonság átdefiniálja az örökölt <*propname*> 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[1], ArrayList osztály[1]	IList interfész[1], List osztály[1]
{unique, ordered}	vector[2]	LinkedHashSet osztály, List interfész[2], ArrayList osztály[2]	IList interfész[2], List osztály[2]
{nonunique, ordered} vagy {sequence} vagy {seq}	vector	List interfész, ArrayList osztály	IList interfész, List osztály

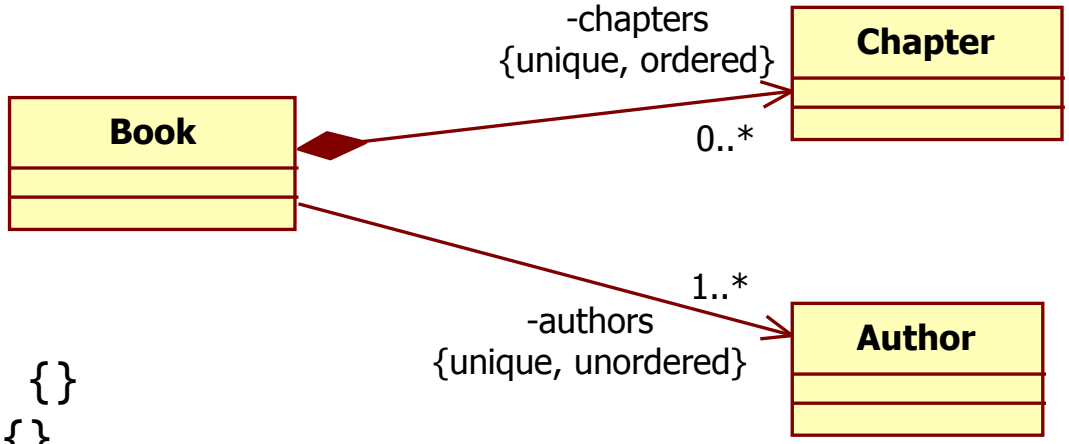
Egyedi (unique) kollekciók egy értéket csak egyszer tárolhatnak.

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

[1] Nem használjuk ki a rendezettséget

[2] Az elemek egyediségét nekünk kell biztosítani

Tulajdonság módosítók példa



C# leképzés:

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

Java leképzés:

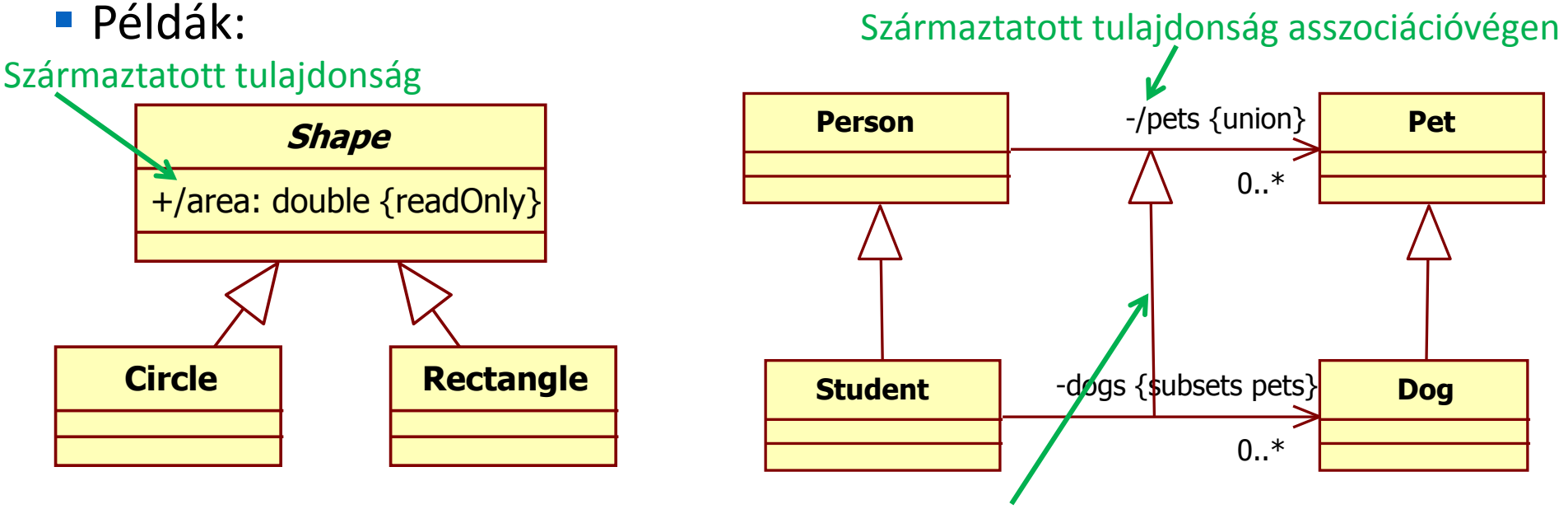
```
public class Chapter {}
public class Author {}
public class Book {
  private ArrayList<Chapter> chapters;
  private HashSet<Author> authors;
}
```

C++ leképzés:

```
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:

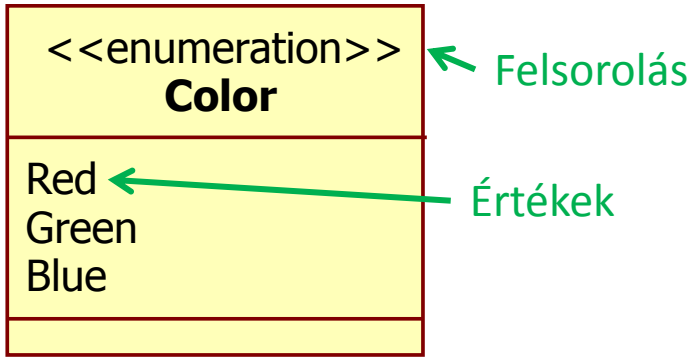


(Asszociációk között is értelmezett az öröklődés.)

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

Felsorolás (enumeration)

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

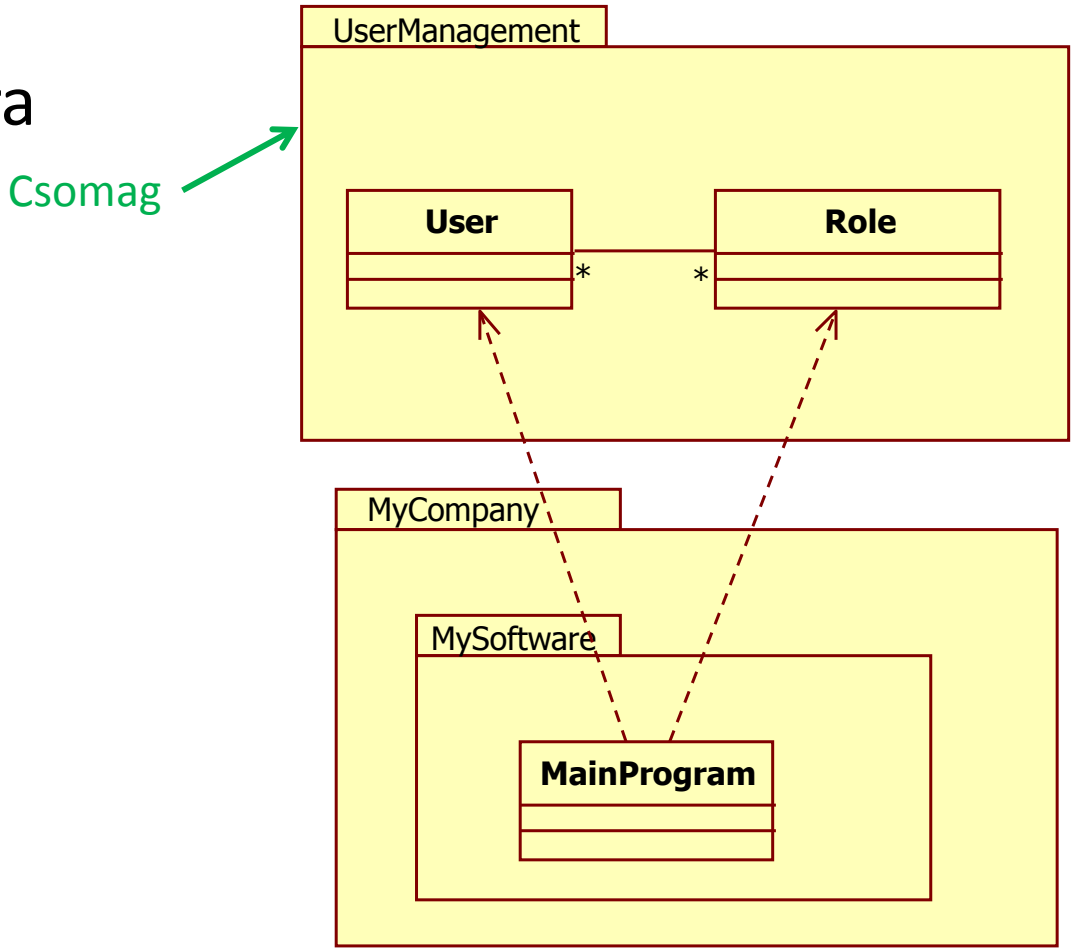


C++/Java/C# leképzés:

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

Csomag (package)

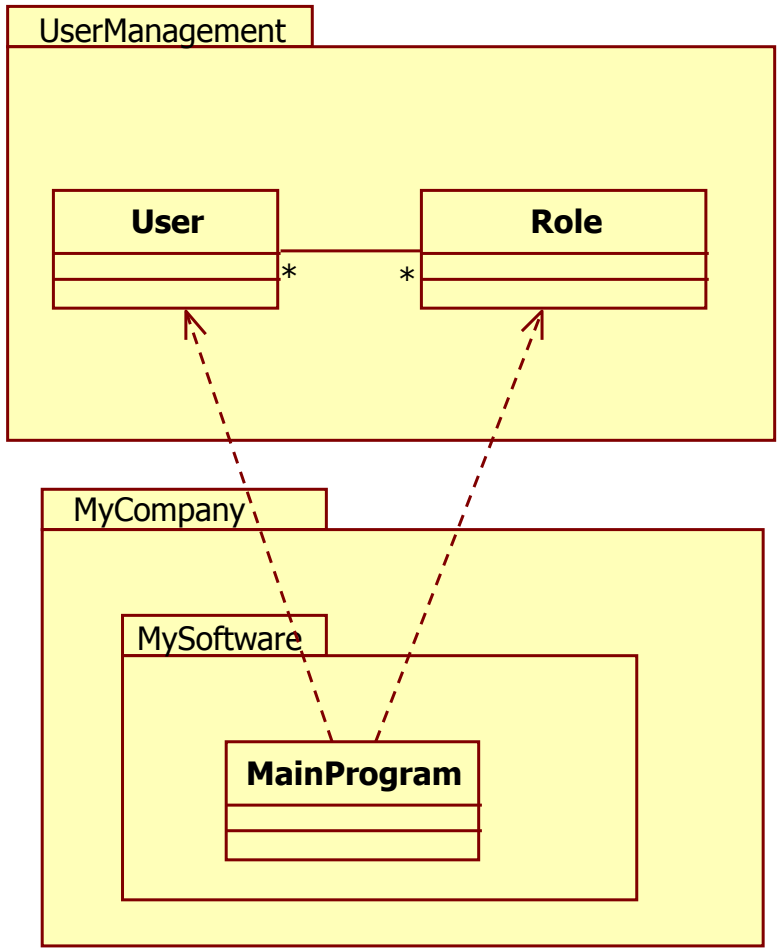
- **Csomag**: különböző elemek csoportosítására szolgál
- Tipikus programnyelvi leképzés: csomag vagy névter



Csomag (package)

C++ leképzés:

```
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:

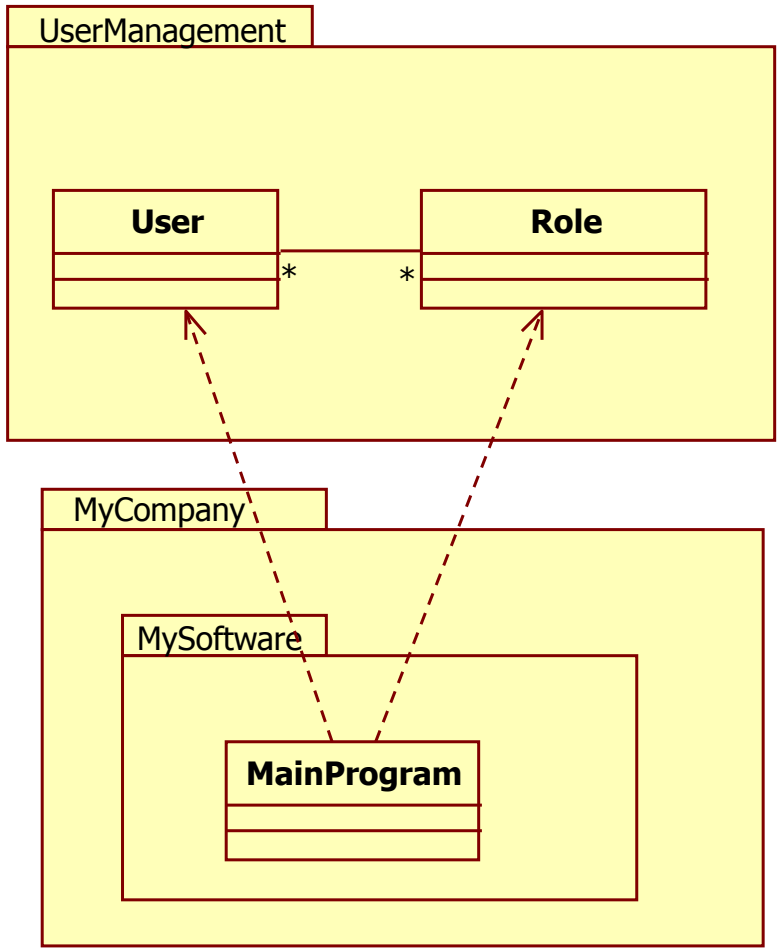
```
package usermanagement;
import java.util.HashSet;
public class User {
  private HashSet<Role> roles;
}
```

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 {
}
```



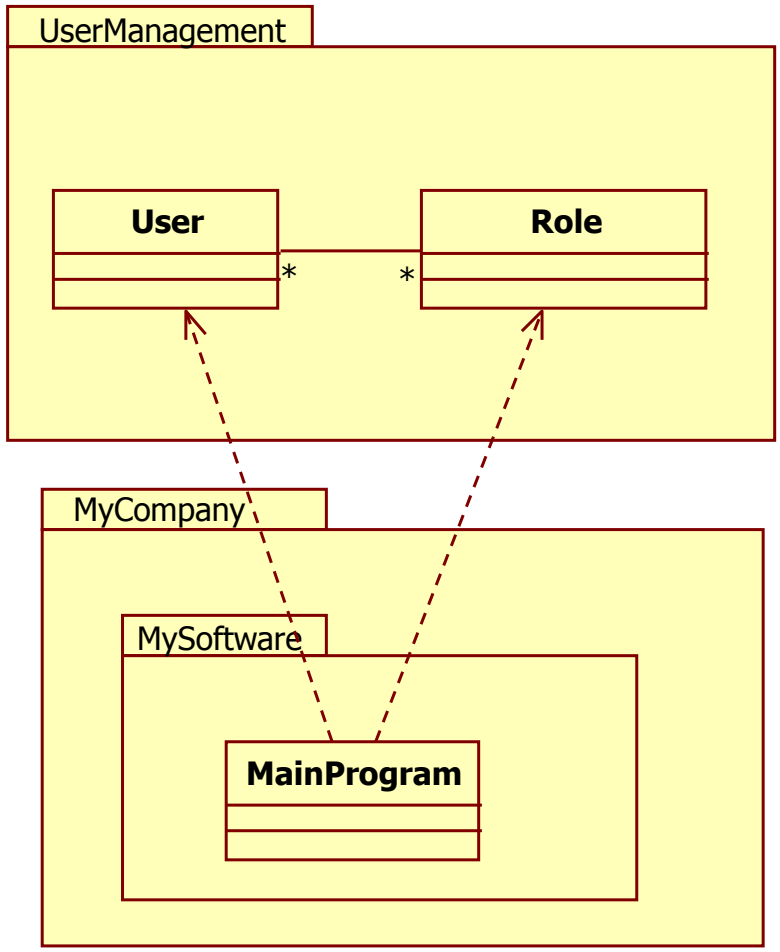
Csomag (package)

C# leképzés:

```
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 diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

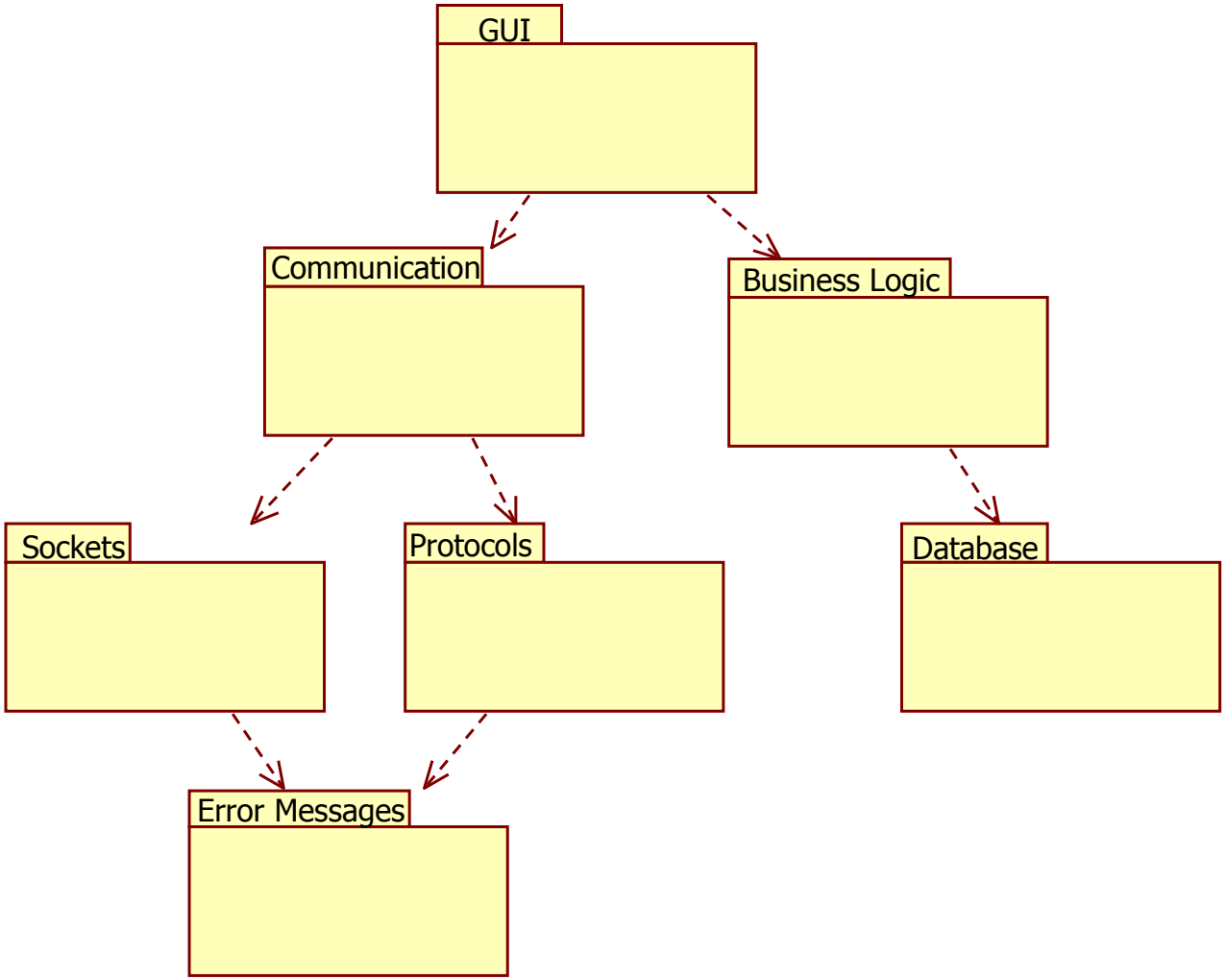
Viselkedési UML diagramok:

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

Csomagdiagram (Package Diagram)

Csomagdiagram (Package Diagram)

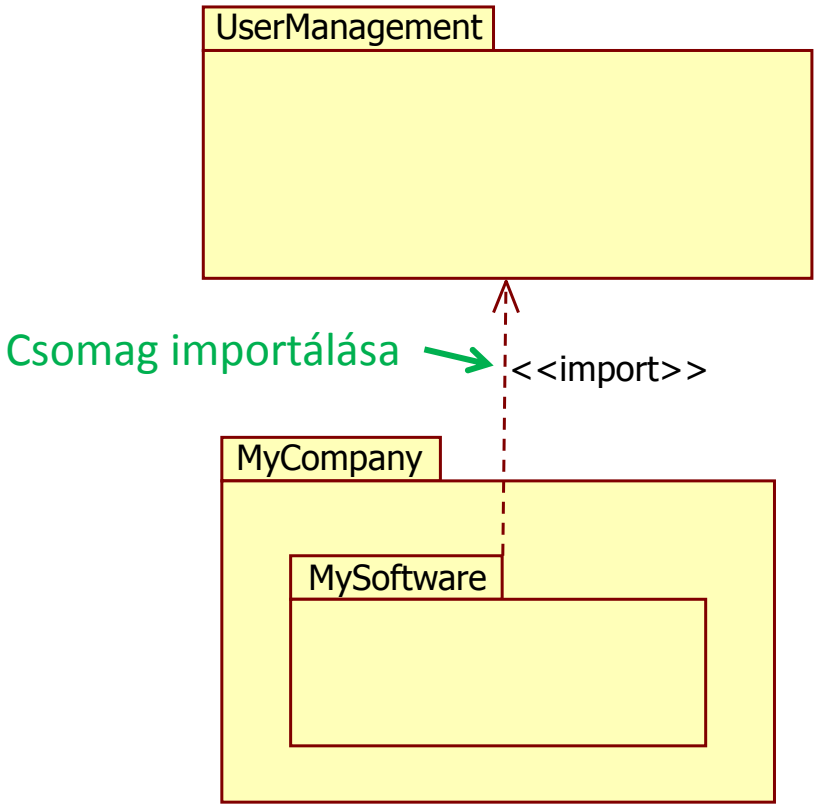
- A csomagdiagram csomagok közötti **függőségeket (dependencies)** ábrázol
- Példa:



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

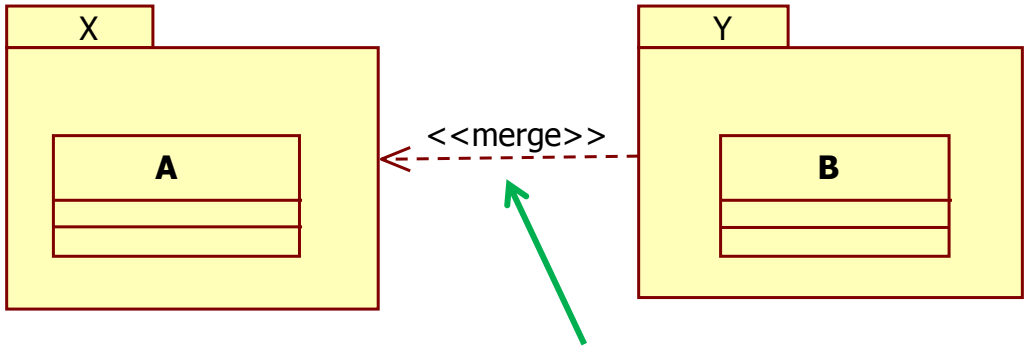
Csomag importálása (Package import)

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



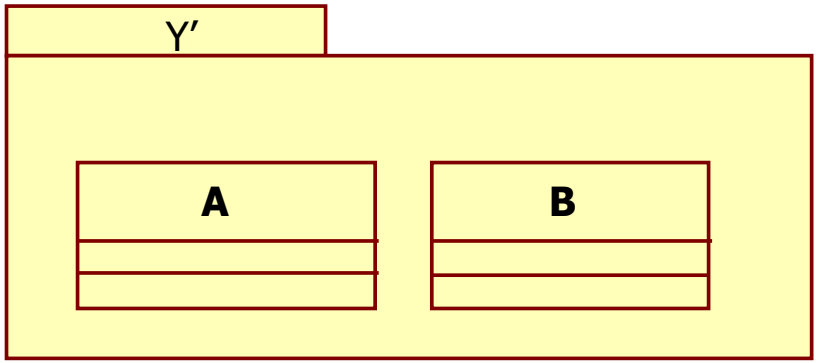
Csomagok összefésülése (Package merge)

- 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 diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

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

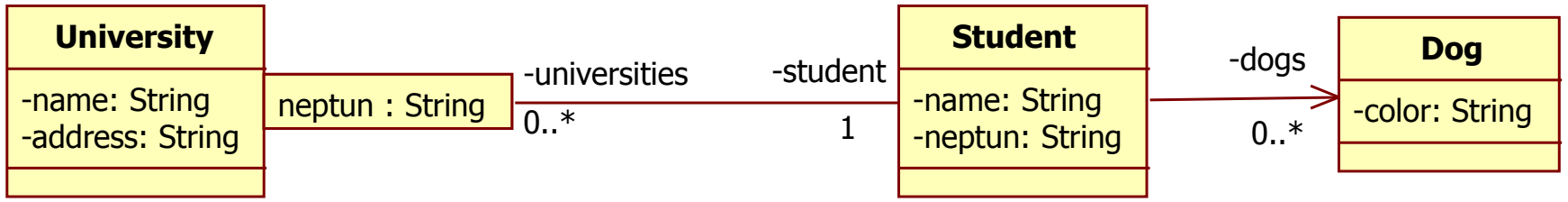
Objektumdiagram (Object Diagram)

Objektumdiagram (Object Diagram)

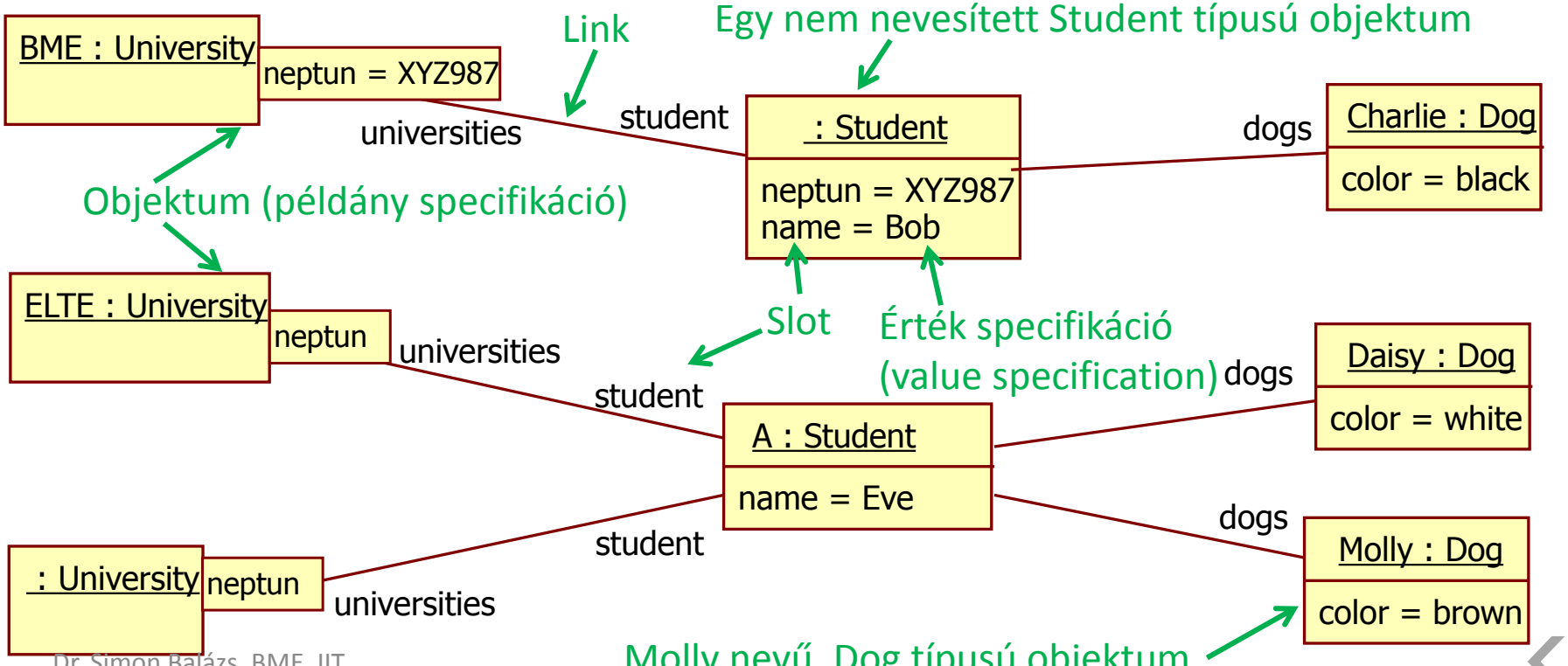
- Az objektumdiagram egy példányokból álló gráf (a csomópontok objektumok és értékek)
- Az objektumdiagram az osztálydiagram egy példánya:
  - osztályok példányai: **objektum (object)** vagy **példány specifikáció (instance specification)**
  - asszociációk példányai: **link**
- A rendszer részletes 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 diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

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

Köszönöm a figyelmet!

Unified Modeling Language

Szoftvertechnológia

Dr. Goldschmidt Balázs

BME, IIT

Tartalom

- UML diagramok:
  - Szekvenciadiagram
  - Kommunikációs diagram
  - Interakciós áttekintő diagram

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

Hol tartunk?

Osztály-diagram ← Egy komponens/rendszer struktúrája objektumorientált modellként

Csomag-diagram ← Csomagok és a közöttük lévő függőségek

Objektum-diagram ← A rendszer részletes állapotának ábrázolása egy adott időpillanatban

Most következik:
Hogyan tervezzük meg egy komponens belsejét?
(Viselkedés)

Hol tartunk?

Most következik:
Hogyan tervezzük meg egy komponens belsejét?
(Viselkedés)

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

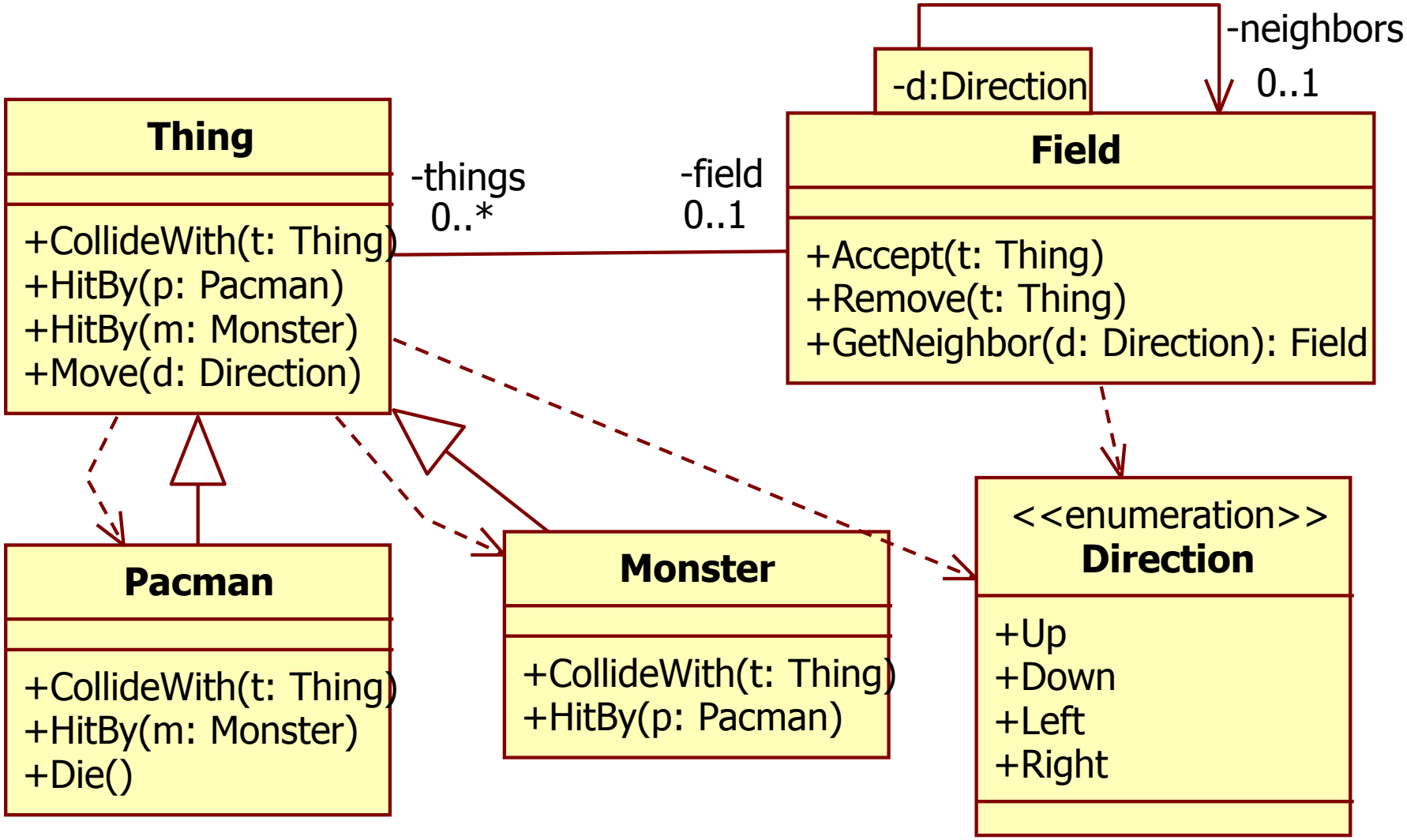
Use case diagram	Aktivitásdiagram	Szekvenciadiagram	Kommunikációs diagram
Állapotdiagram	Időzítődiagram	Interakciós áttekintő diagram	

Szekvenciadiagram (Sequence Diagram)

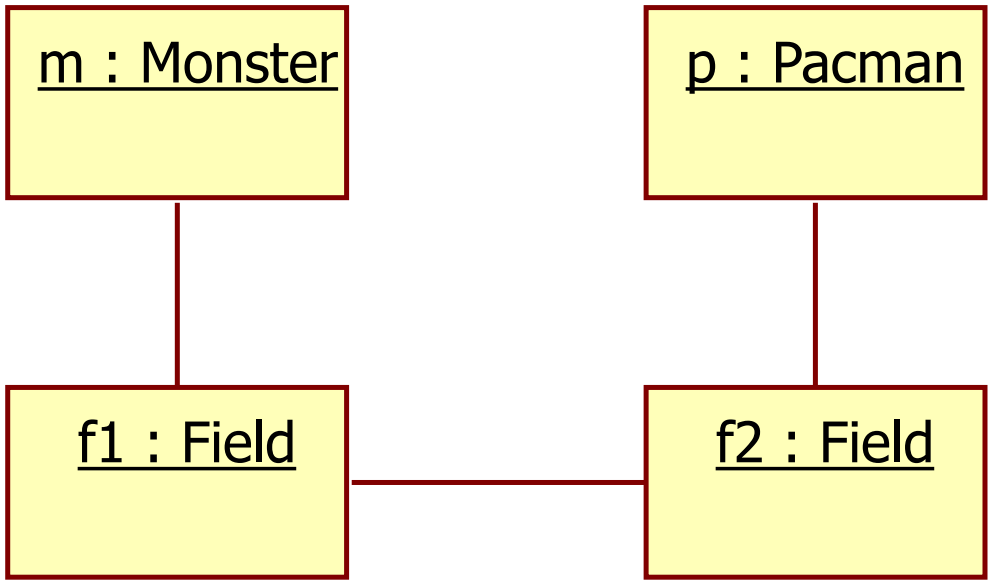
Szekvenciadiagram (Sequence Diagram)

- Interakciók grafikus ábrázolására szolgál
  - a rendszer dinamikus viselkedését mutatja
  - az interakciók a részvevők közötti információcserére fókuszálnak
- A szekvenciadiagram használható use case forgatókönyvek leírására, metódusok belső logikájának definiálására és egy protokollban történő üzenetváltások ábrázolására
- Egy szekvenciadiagram üzenetváltások egy lehetséges időbeli lefutását mutatja
  - egyszerű futássorozatok vannak ábrázolva
  - le lehet írni helyes és helytelen lefutásokat is
  - a nem ábrázolt lefutásokról nem mindig dönthető el egyértelműen, hogy helyesek vagy helytelenek

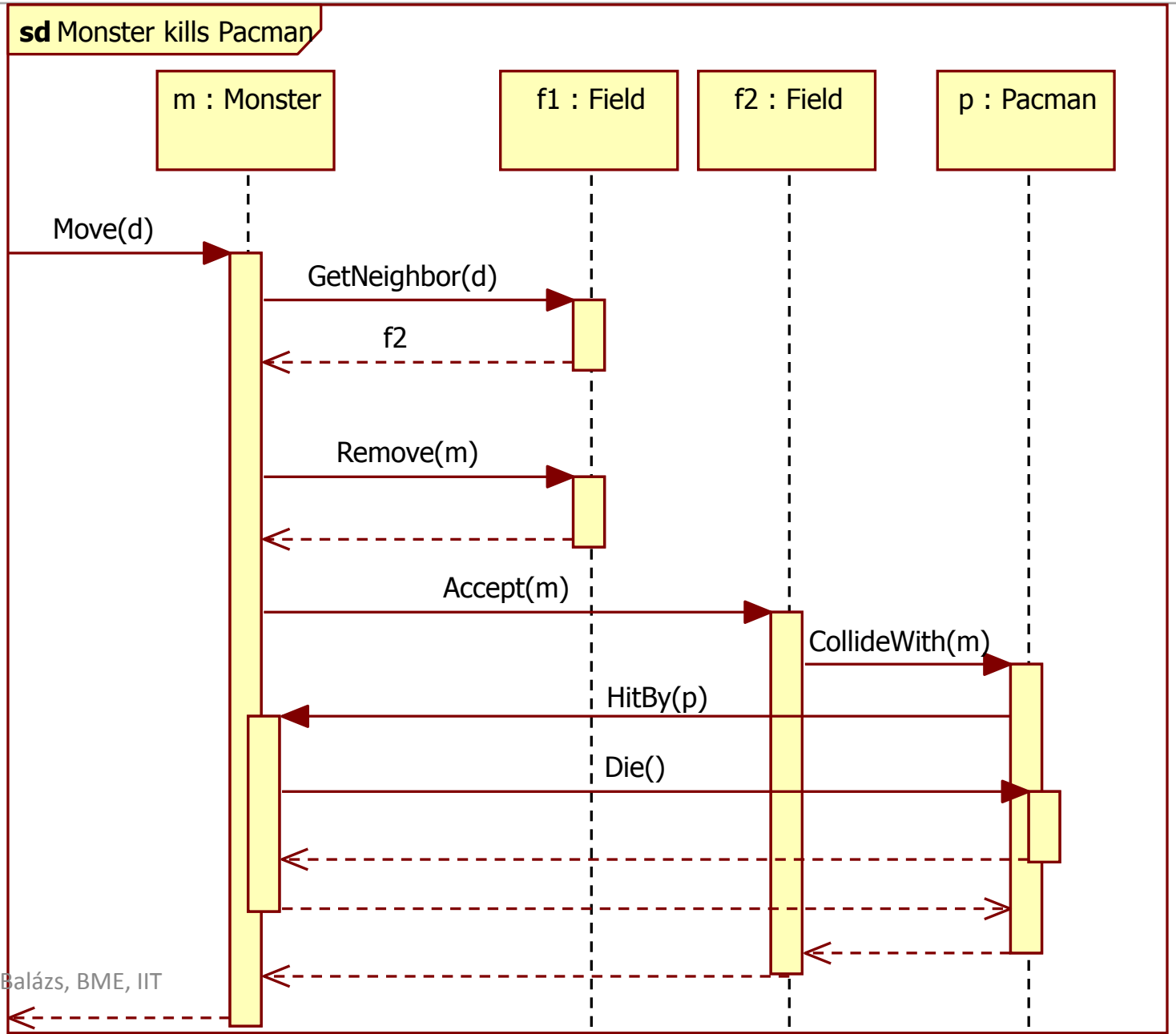
Pacman: osztálydiagram részlet



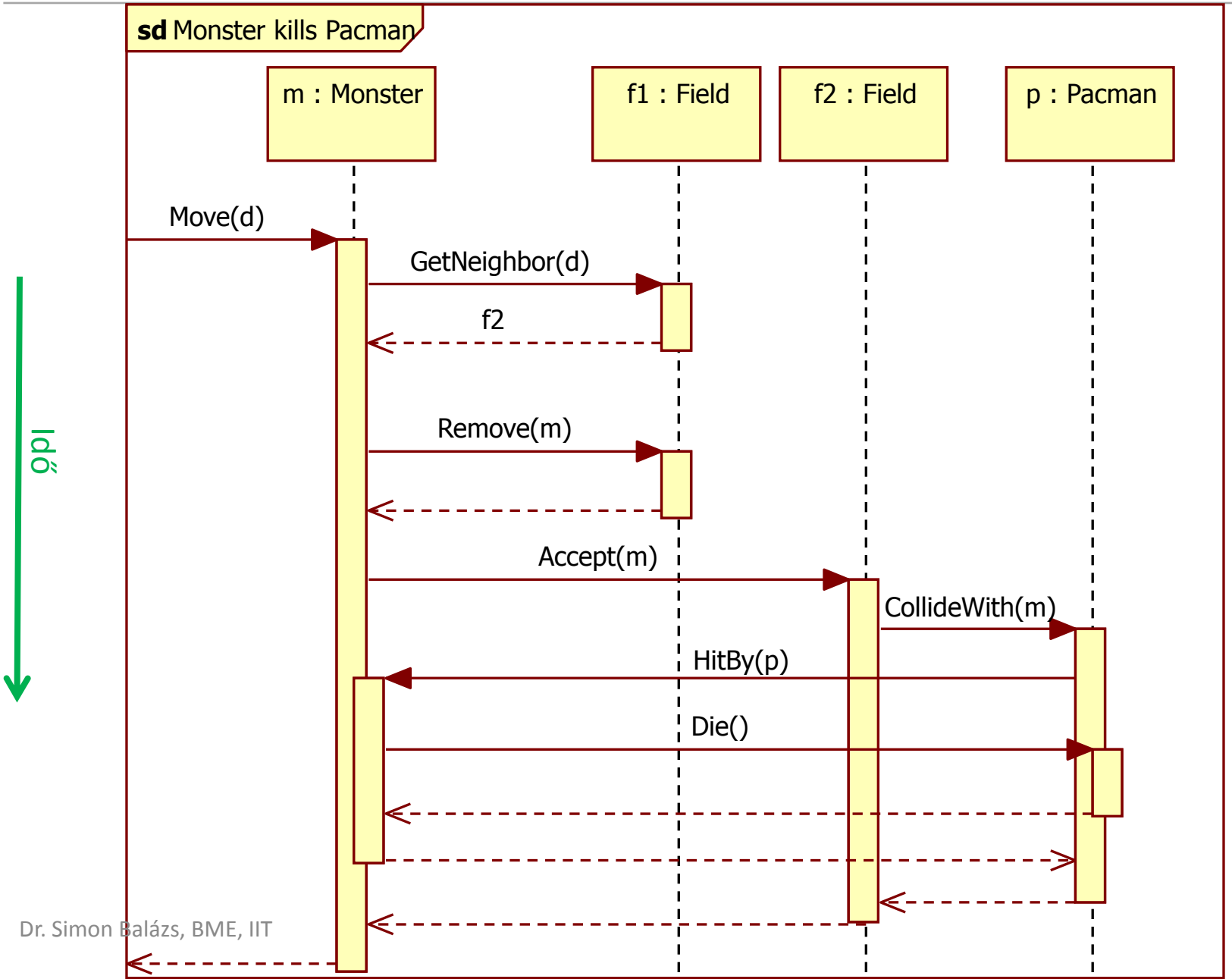
Pacman: objektumdiagram egy lehetséges kezdőállapotra



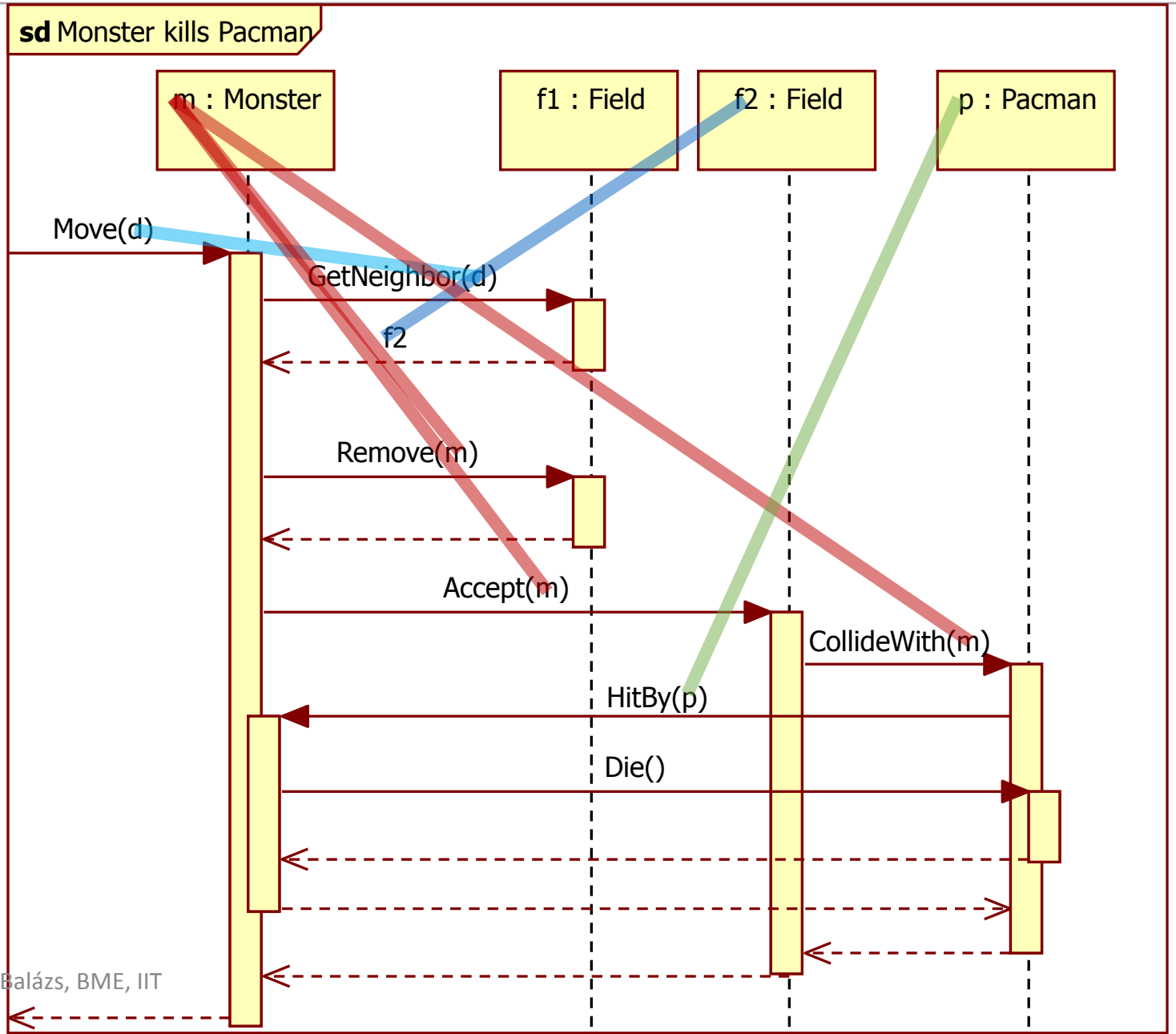
Szekvenciadiagram: a szörny megeszi a Pacmant



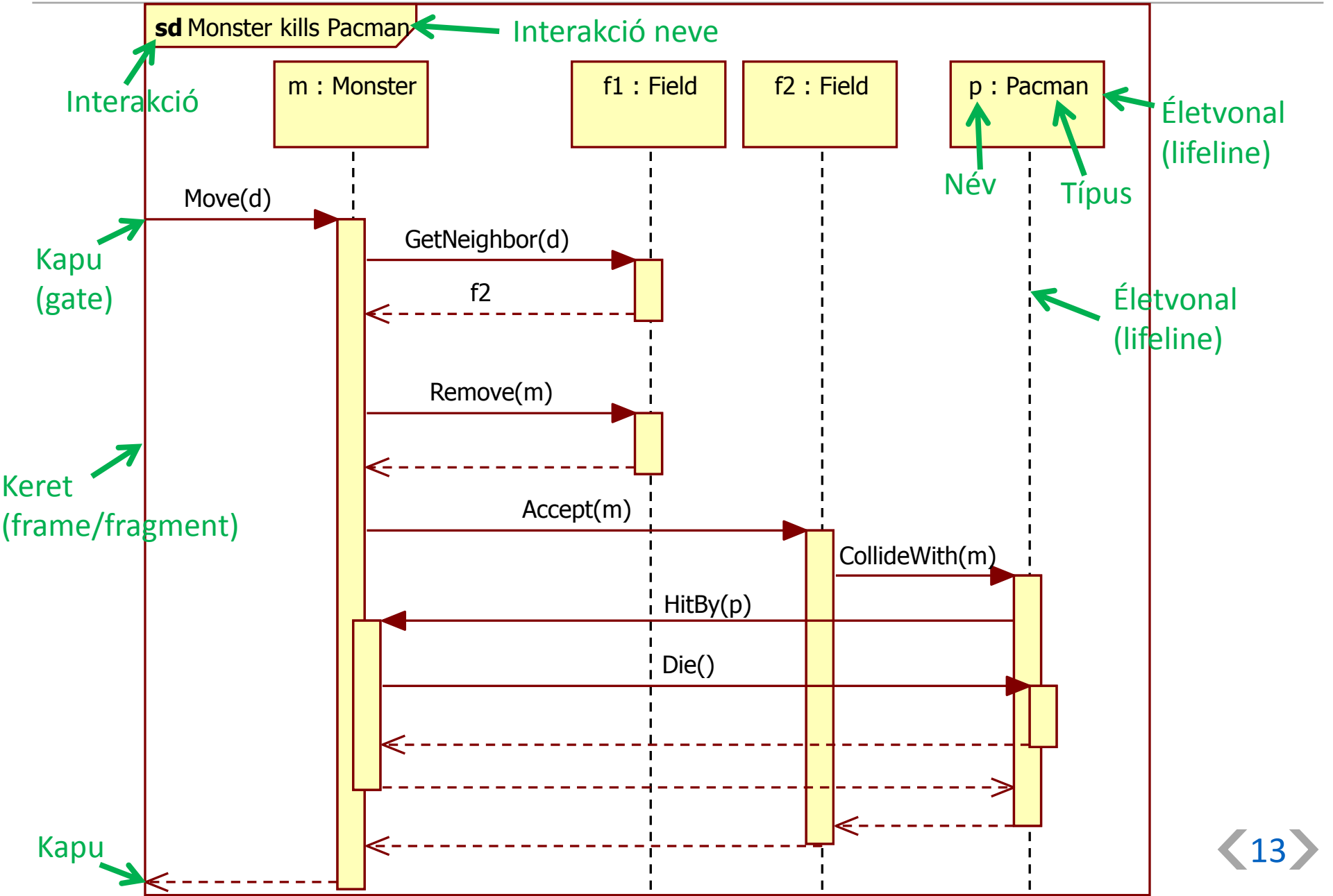
Szekvenciadiagram: a szörny megeszi a Pacmant



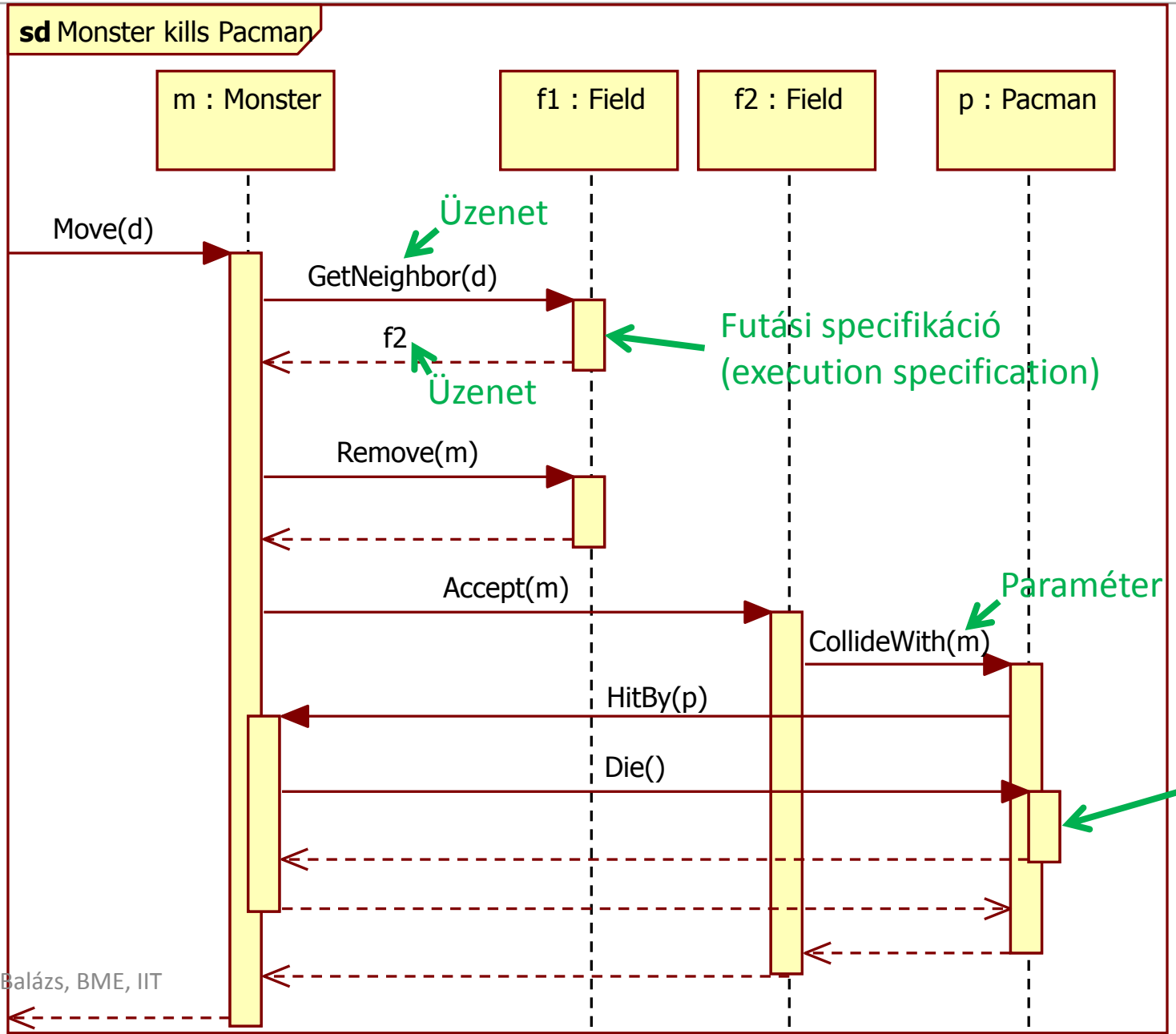
Szekvenciadiagram: a szörny megeszi a Pacmant



Szekvenciadiagram: a szörny megeszi a Pacmant



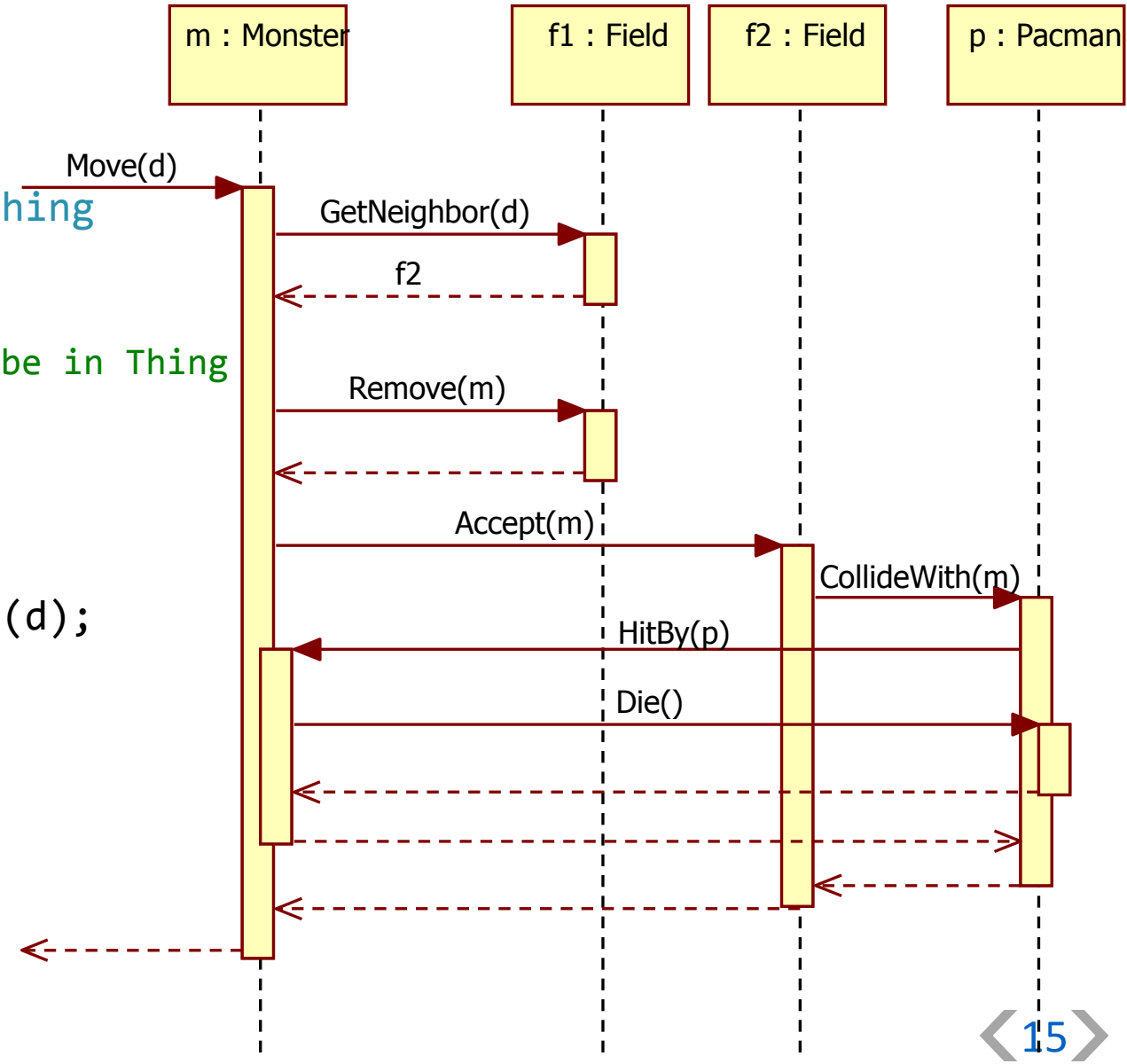
Szekvenciadiagram: a szörny megeszi a Pacmant



Szekvenciadiagram: a szörny megeszi a Pacmant

C++ leképzés:

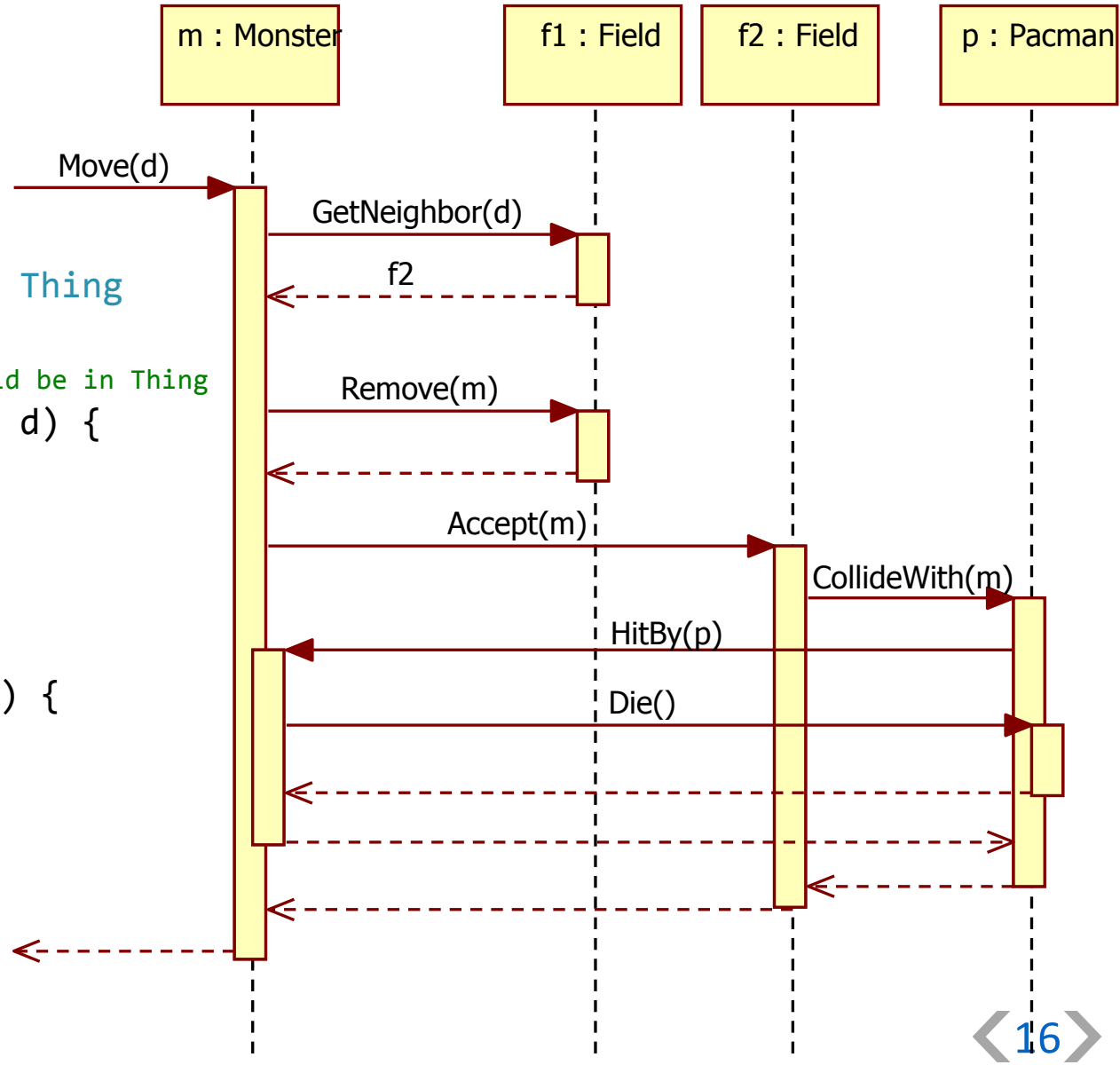
```
class Monster : public Thing
{
private:
  Field* field; //should be in Thing
public:
  void Move(Direction d)
  {
    Field* next =
      field->GetNeighbor(d);
    field->Remove(this);
    next->Accept(this);
  }
  void HitBy(Pacman* p)
  {
    p->Die();
  }
  // ...
}
```



Szekvenciadiagram: a szörny megeszi a Pacmant

Java leképzés:

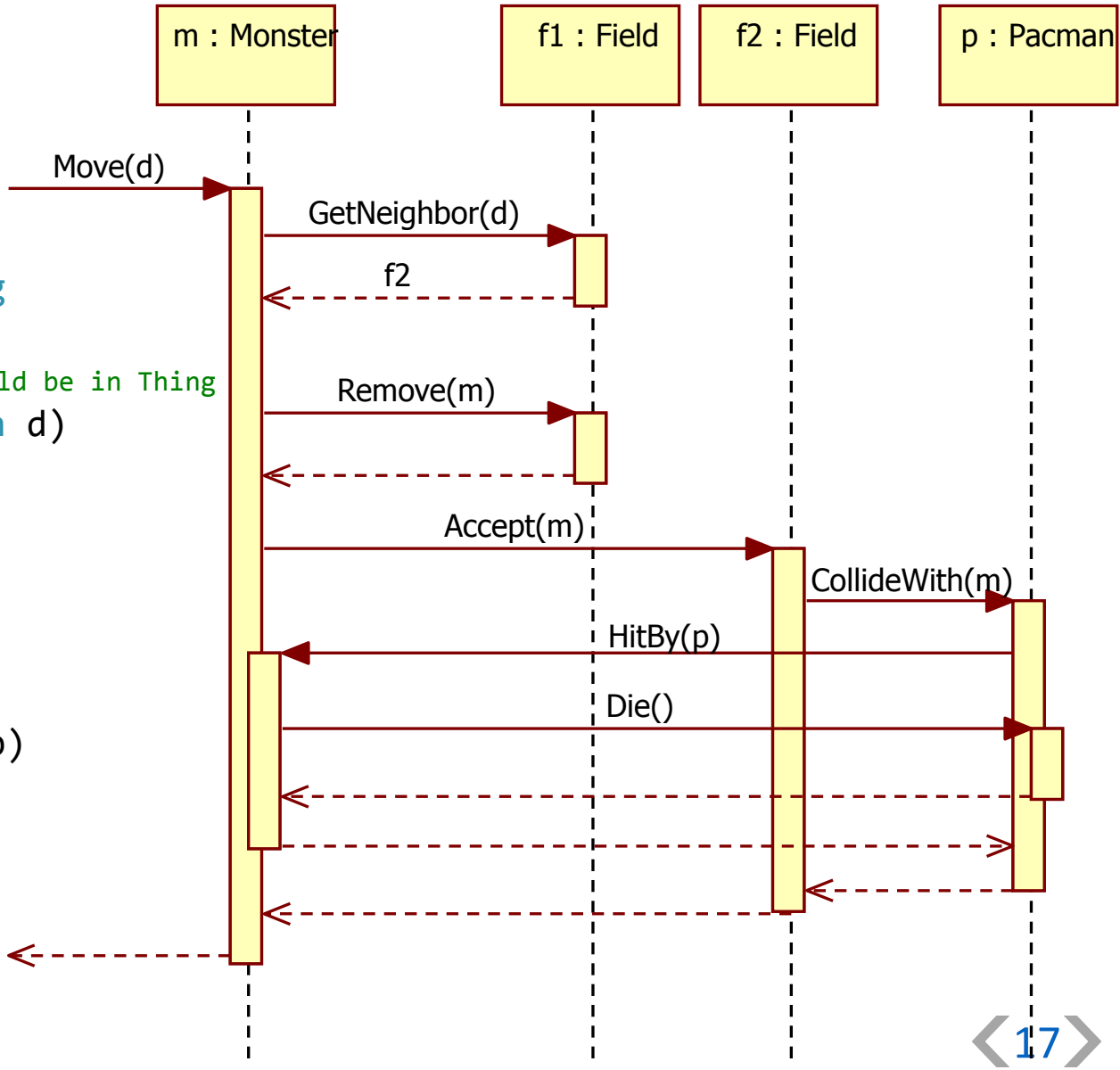
```
public class Monster extends Thing
{
  private Field field; //should be in Thing
  public void Move(Direction d) {
    Field next =
      field.GetNeighbor(d);
    field.Remove(this);
    next.Accept(this);
  }
  public void HitBy(Pacman p) {
    p.Die();
  }
  // ...
}
```



Szekvenciadiagram: a szörny megeszi a Pacmant

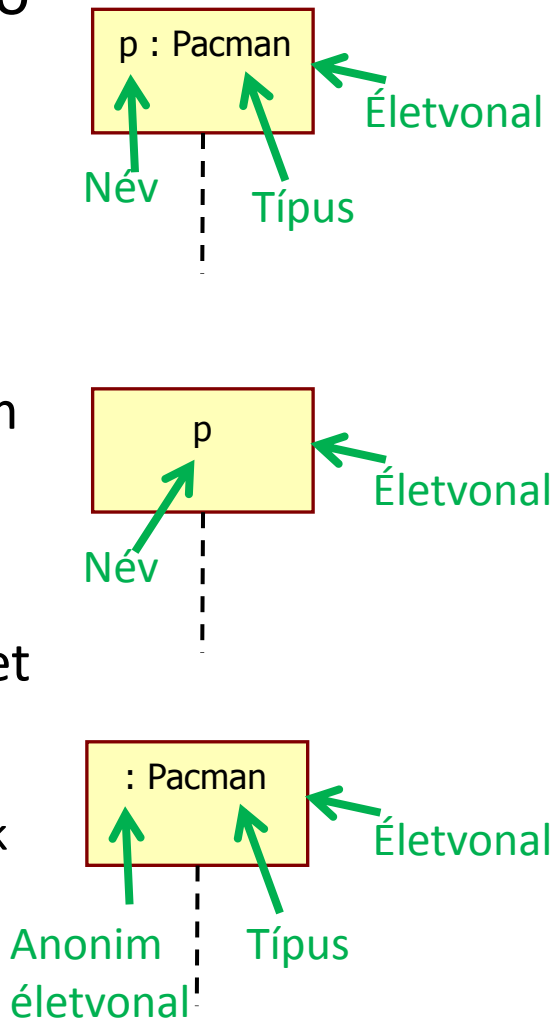
C# leképzés:

```
public class Monster : Thing
{
  private Field field; //should be in Thing
  public void Move(Direction d)
  {
    Field next =
      field.GetNeighbor(d);
    field.Remove(this);
    next.Accept(this);
  }
  public void HitBy(Pacman p)
  {
    p.Die();
  }
  // ...
}
```



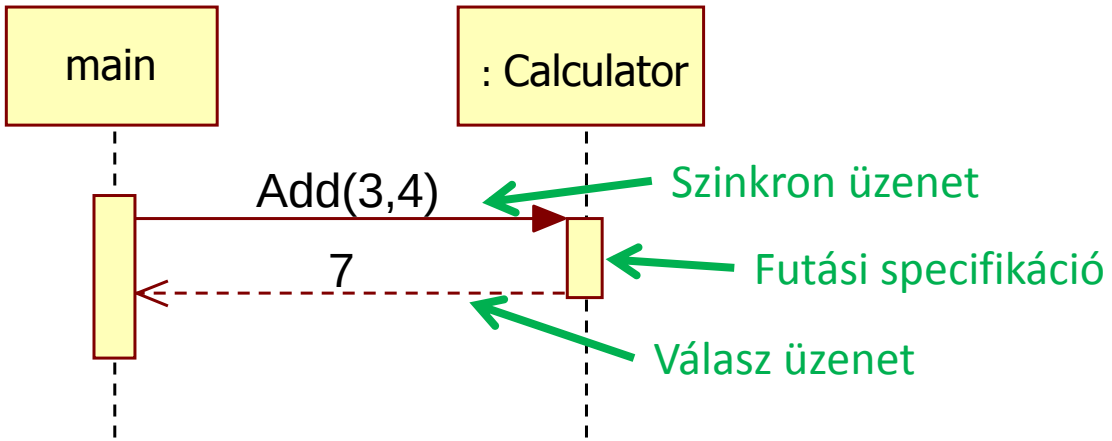
Életvonal (lifeline)

- Egy folyamat idővonalát ábrázolja, ahol az idő fentről lefelé telik
- Az életvonal feje nem objektum!
  - egy objektum (példány specifikáció) az csak egy pillanatkép a dinamikus példányról, amit ő modellez
  - a szekvenciadiagram nem egy pillanatkép, hanem egy időbeli folyamat
  - vagyis: a lifeline fejében *nem kell aláhúzni* a szöveget
  - akár interfész vagy absztrakt osztály is szerepelhet a lifeline fejében
    - de természetesen absztrakt függvények vagy interfészek függvényei csak hívhatók, de mivel nekik nincs implementációjuk, ők nem hívhatnak más függvényeket
- A név és a típus is opcionális, de legalább egyiknek szerepelnie kell



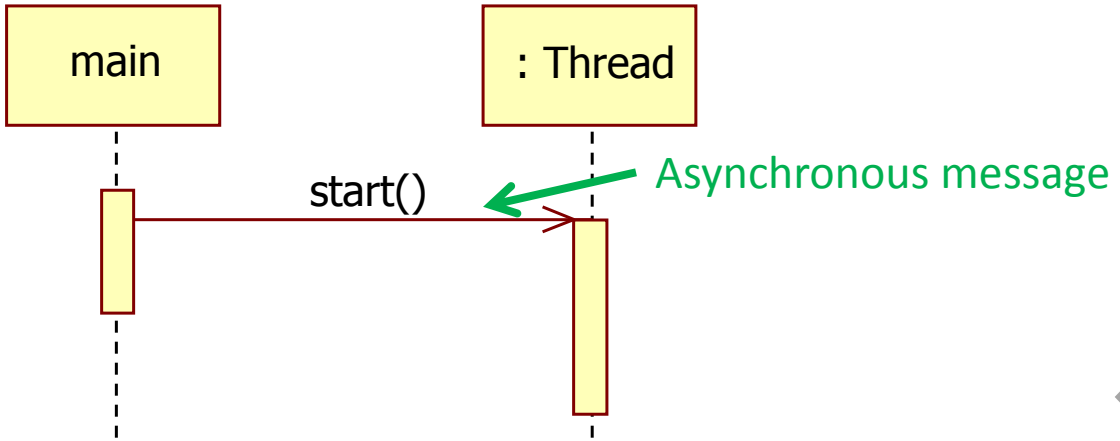
Szekvenciadiagram: szinkron üzenet

- Szinkron üzenet (synchronous message):
  - szinkron függvényhívás
  - az operáció neve és a paraméterek a nyíl felett szerepelnek
  - hatására egy futási specifikáció (execution specification) indul
  - a hívó megvárja, amíg a meghívott függvény véget ér
- Válasz üzenet:
  - a futási specifikáció végétől van rajzolva
  - a visszatérési érték a szaggatott vonal felett szerepel
- Futási specifikáció (execution specification):
  - egy függvényhívás törzsének futását jelöli, ekkor aktív a függvény
  - a programozási nyelvekben ez a stack frame



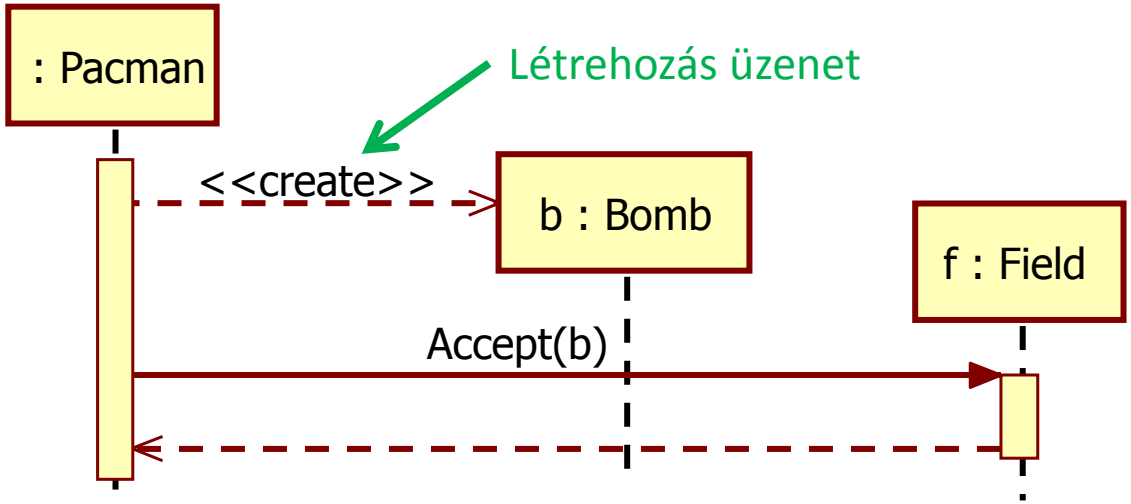
Szekvenciadiagram: aszinkron üzenet

- Aszinkron üzenet (asynchronous message), jelzés (signal):
  - aszinkron függvényhívás
  - a hívó nem várja meg a függvény lefutásának végét
    - egy másik szálat vagy folyamatot indít el
  - az operáció neve és a paraméterek a nyíl felett szerepelnek
  - hatására egy futási specifikáció (execution specification) indul
    - ami nem feltétlenül fér bele a hívó futási specifikációjának időtartamába
  - aszinkron üzenetnek nincs válaszüzenete
    - egy másik aszinkron hívással lehet visszahívást (callback) végezni

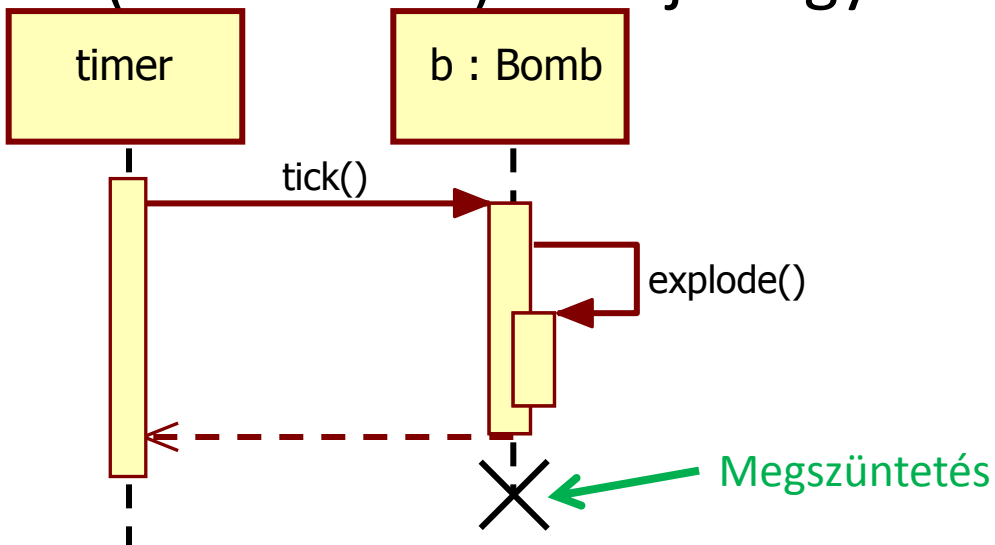


Szekvenciadiagram: létrehozás és megszüntetés

- Létrehozás üzenet (create message): új életvonalat készít



- Megszüntetés (destruction): befejez egy életvonalat



Szekvenciadiagram: alternatívák (alternatives)

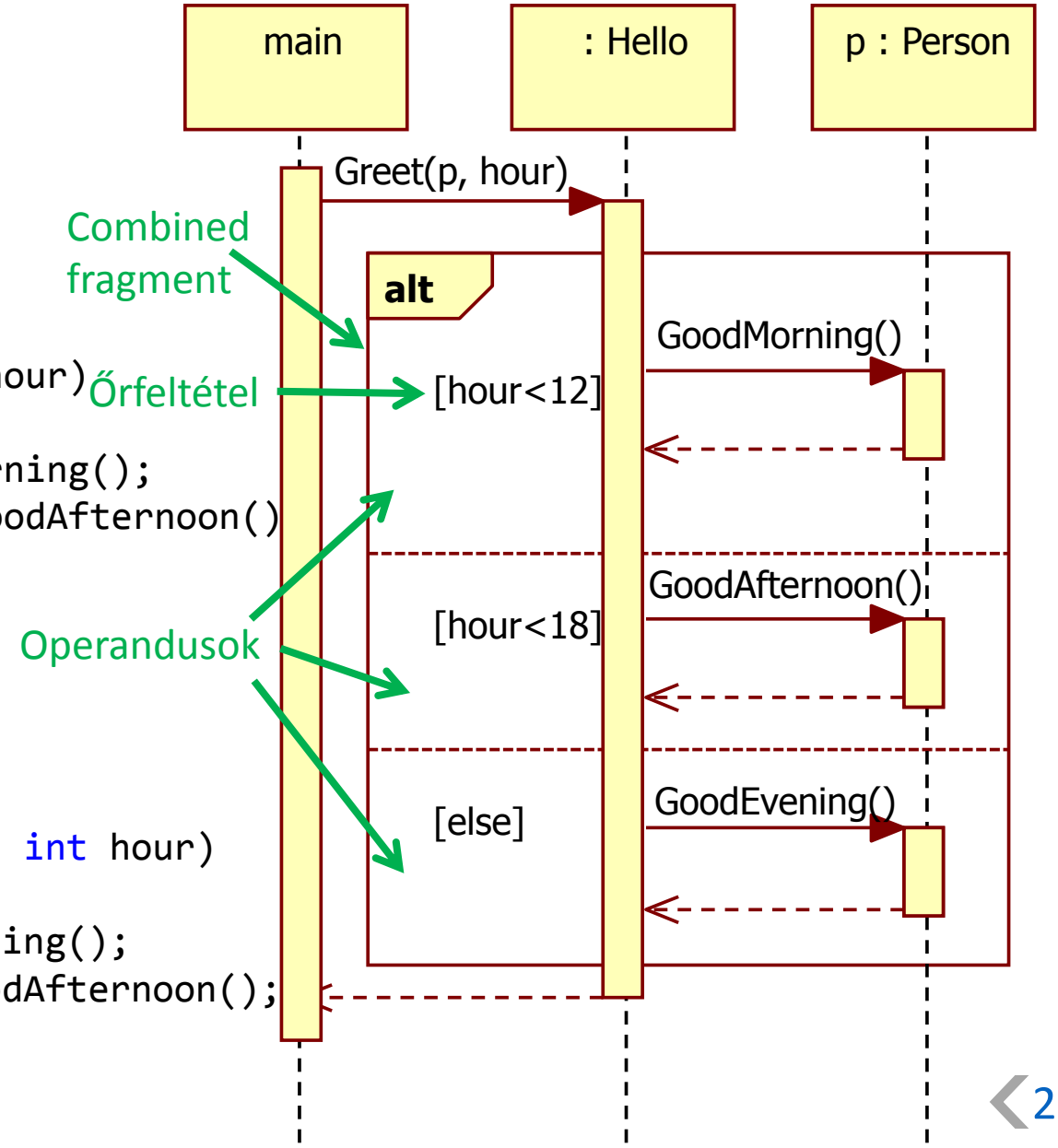
- Alternatívák: **alt**
  - if-else ágak

C++ leképzés:

```
class Hello {
public:
  void Greet(Person* p, int hour)
  {
    if (hour < 12) p->GoodMorning();
    else if (hour < 18) p->GoodAfternoon();
    else p->GoodEvening();
  }
}
```

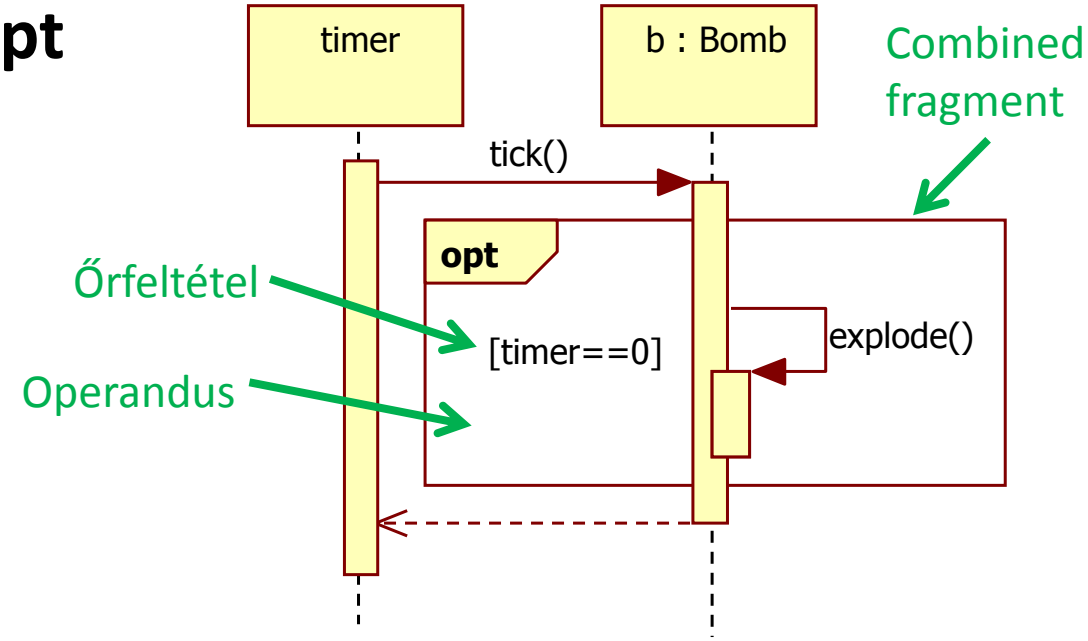
Java/C# leképzés:

```
public class Hello {
  public void Greet(Person p, int hour)
  {
    if (hour < 12) p.GoodMorning();
    else if (hour < 18) p.GoodAfternoon();
    else p.GoodEvening();
  }
}
```



Szekvenciadiagram: opció (option)

- Feltételes viselkedés: **opt**
  - else nélküli if



C++ leképzés:

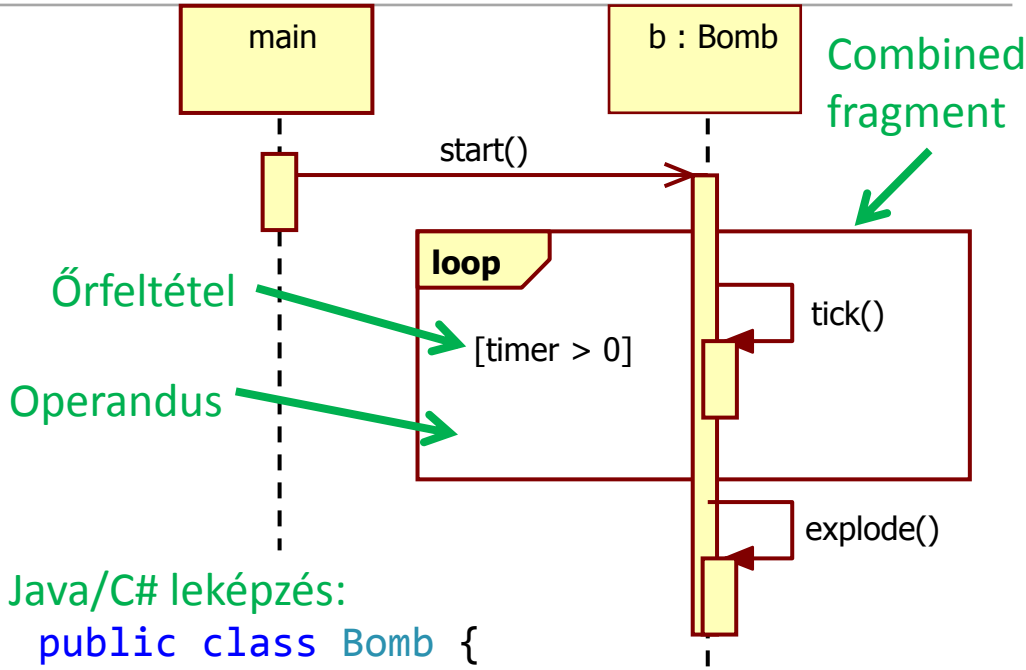
```
class Bomb {
private:
  int timer;
public:
  void Tick() {
    --timer;
    if (timer == 0) {
      this.Explode();
    }
  }
  void Explode() { /*...*/ }
}
```

Java/C# leképzés:

```
public class Bomb {
  private int timer;
  public void Tick() {
    --timer;
    if (timer == 0) {
      this.Explode();
    }
  }
  public void Explode() { /*...*/ }
}
```

Szekvenciadiagram: ciklus (loop)

- Ismételt viselkedés: **loop**



C++ leképzés:

```
class Bomb
{
private:
  int timer;
public:
  void Start()
  {
    while(timer > 0)
    {
      this.Tick();
    }
    this.Explode();
  }
  void Tick()
  {
    --timer;
  }
  void Explode() { /*...*/ }
}
```

Java/C# leképzés:

```
public class Bomb {
  private int timer;
  public void Start() {
    while(timer > 0) {
      this.Tick();
    }
    this.Explode();
  }
  public void Tick() {
    --timer;
  }
  public void Explode() { /*...*/ }
}
```

Szekvenciadiagram : combined fragments

Rövidí-tés	Fajta	Jelentés
alt	Anternatívák	Viselkedés kiválasztása feltétel alapján: legfeljebb egy operandus lesz lefuttatva.
opt	Opció	Opcionális viselkedés feltétel alapján: vagy lefut az egyetlen operandus, vagy nem történik semmi.
break	Megszakítás	A tartalmazó fragment futása megszakad, és a maradék rész nem fut le.
par	Párhuzamos	Párhuzamosan futnak le az operandusok.
seq	Gyenge sorrend	Gyenge sorrendet határoz meg az operandusok között: csak az azonos lifeline-on belül kell tartani a sorrendet.
strict	Erős sorrend	Erős sorrendet határoz meg az operandusok között: a függőleges koordináta szigorúan meghatározza a sorrendet.

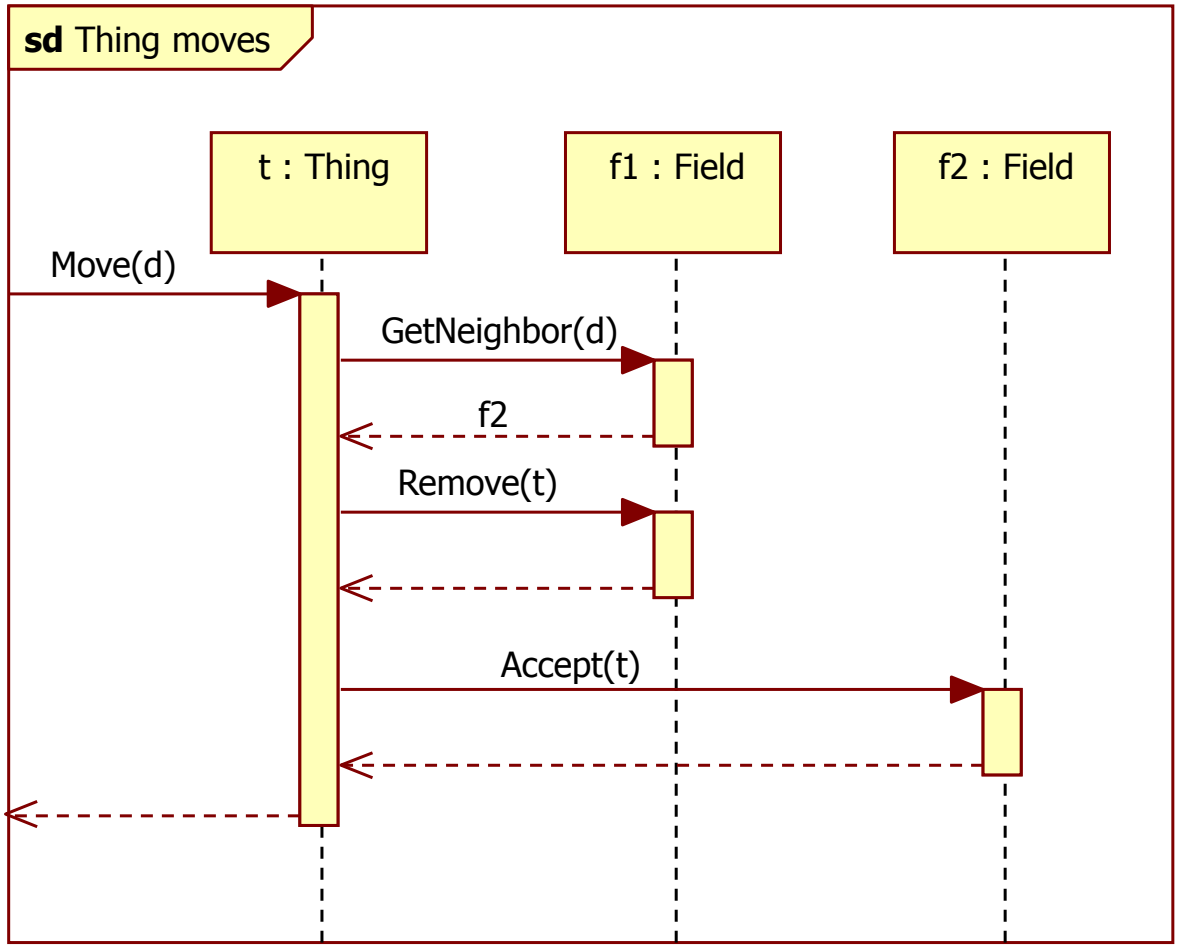
Szekvenciadiagram : combined fragments

Rövidí-tés	Fajta	Jelentés
neg	Negatív	Helytelen lefutásokat mutat. Minden ezektől eltérő lefutás helyesnek és lehetségesnek számít.
critical	Kritikus szakasz	Atomi műveletnek tekinthető a tartalmazó fragment szempontjából.
ignore	Rejtett üzenetek	Azt jelzi, hogy néhány üzenet nincs megmutatva az adott fragmentben.
consider	Fontos üzenetek	Azt jelzi, hogy néhány üzenet fontos az adott fragmentben, a többi üzenet rejtettnek tekinthető.
assert	Állítás	Egy állítást reprezentál. Csak az operandus szekvenciái a helyes folytatások, minden más folytatás helytelen.
loop	Ciklus	Ciklust reprezentál: az operandus ismételten le lesz futtatva.

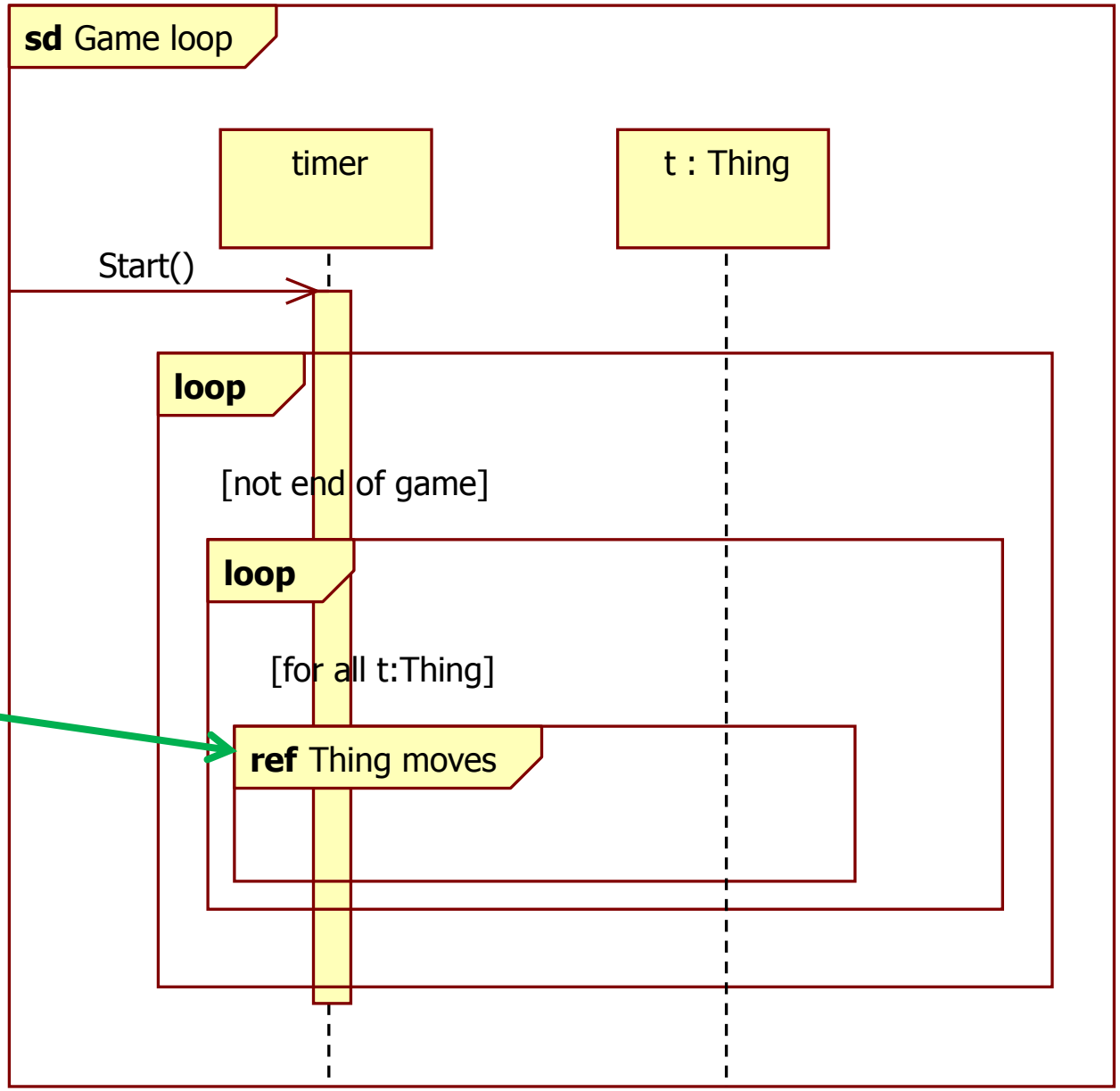
Szekvenciadiagram : interakció használata (interaction use)

- A szekvenciadiagramok meghivatkozhatnak más szekvenciadiagramokat az interakció használata segítségével
- Előnyök:
  - modularitás: nagy diagramok feldarabolhatók több kisebb diagramra
  - újrahasznosítás: több diagramon előforduló azonos viselkedés kiemelhető és meghivatkozható egy közös diagram alapján
  - olvashatóság: magasabb szintű diagramok finomíthatók alacsonyabb szintű diagramok segítségével

Szekvenciadiagram: Thing moves



Szekvenciadiagram: Interakció használata



Interakció használata

(meghivatkozza az előző dián szereplő szekvenciát)

Konzisztens és kezelhető szekvenciadiagramok

- Egy diagram pontosan egy viselkedést ábrázoljon
- Azonos típusú, de különböző objektumok külön lifeline-t kapjanak
- A hívónak ismernie kell a cél objektumot
  - egy objektumdiagramon ábrázolhatjuk a kezdeti ismeretségeket
- A hívott függvénynek elérhetőnek kell lennie a használt objektum ismert statikus típusa alapján
- A polimorfikus viselkedés leírására külön diagramokat rajzoljunk
- Ugyanaz a függvény ugyanúgy viselkedjen különböző diagramokon
- Tüntessük fel a paramétereket és a visszatérési értékeket is
- Az operációknak léteznie kell az osztálydiagramon

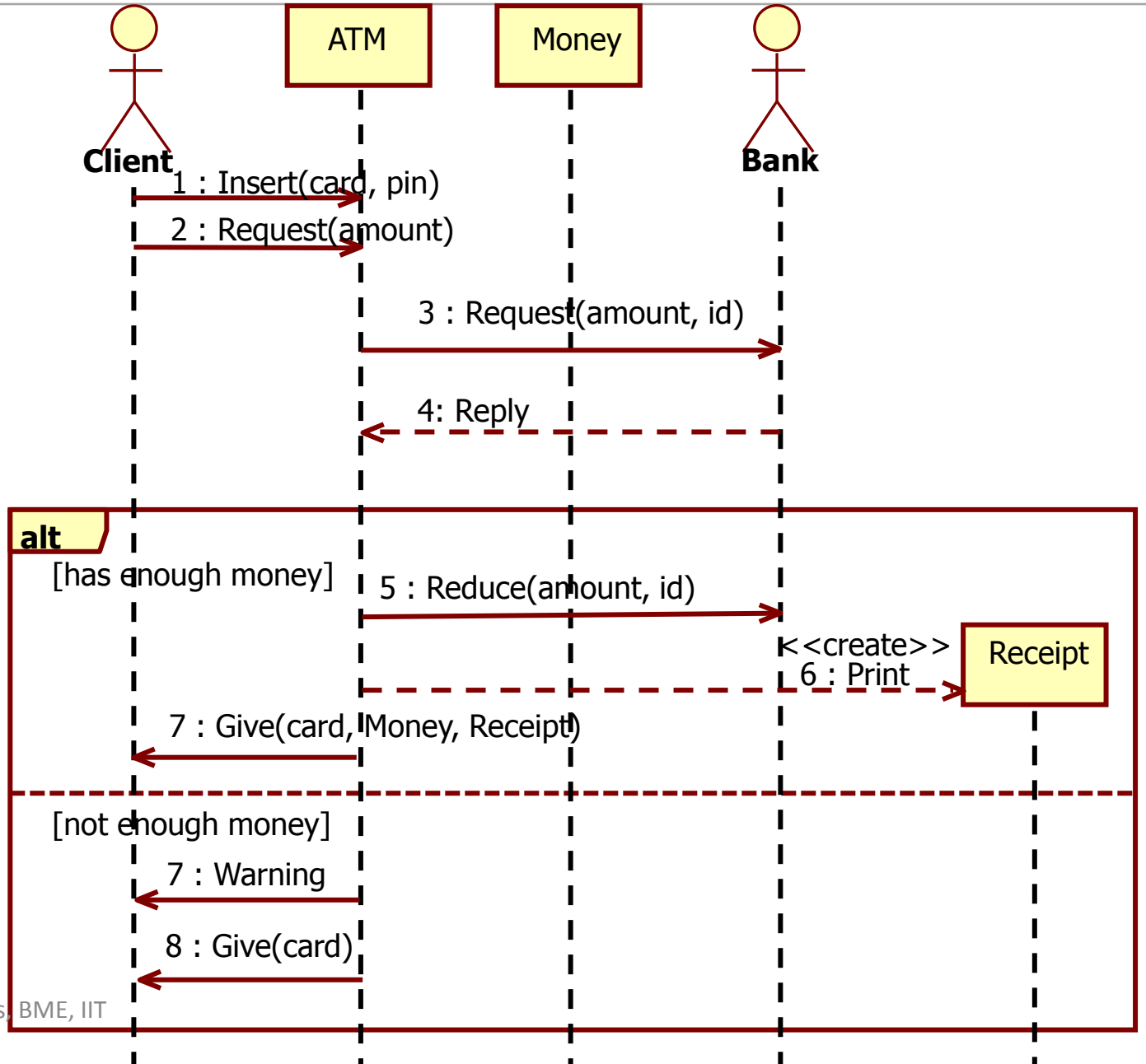
Informális szekvenciadiagram

- Informális szekvenciadiagramok:
  - enyhítenek az UML szigorú formalizmusán
  - nem pontosan követik az UML szabványt
  - de kiválóan alkalmasak ötletek felvázolására
- Tipikusan nem rajzoljuk meg a rendszer minden egyes apró részletét formális diagramokon
  - általában ez szükségtelen
  - sokszor időpazarló: a viselkedés leprogramozása egyszerűbb és gyorsabb, mint béna eszközökkel szekvenciadiagramokat rajzolgatni
  - szinkronban tartani őket a forráskóddal is időpazarló, és nehéz
- Általában az informális diagramok elegendő információt adnak a rendszer működéséről
  - ezeket könnyebb használni vázlatként
  - elegendő részletet adnak az áttekintő kép megértéséhez
  - csak akkor kell őket frissíteni a dokumentációban, ha az általuk mutatott viselkedést befolyásoló tervezői döntések megváltoznak
- Megjegyzés: a házi feladatban és a vizsgán, valamint a „Szoftver projekt labor” c. tárgyban részletes ***formális*** diagramokat várunk el
  - a célunk, hogy lássuk, sikerült-e elsajátítani az UML szabványt

Példa: Pénzfelvétel use case

- **Use case:** Pénzfelvétel
- **Aktorok:** Ügyfél, Bank
- **Főforgatókönyv:**
  - 1. Az Ügyfél átadja a bankkártyát és a pinkódot
  - 2. Az ATM ellenőrzi, hogy a bankkártyához tartozik-e a pinkód
  - 3. Az Ügyfél megadja, hogy mennyi pénzt venne fel
  - 4. Az ATM megkérdezi a Bank-ot, hogy ez így rendben van-e
  - 5. A Bank megerősíti, hogy mehet a tranzakció
  - 6. Az ATM kiadja a bankkártyát
  - 7. Az Ügyfél elveszi a bankkártyát
  - 8. Az ATM kinyomtatja a bizonylatot és kiadja a pénzzel együtt
  - 9. Az Ügyfél elveszi a pénzt és a bizonylatot
- **Alternatív forgatókönyv 5.A:**
  - 5.A.1. A Bank jelzi, hogy az Ügyfél számláján nincs elég pénz
  - 5.A.2. Az ATM visszaadja a bankkártyát és hibaüzenetet ír
  - 5.A.3. Az Ügyfél elveszi a bankkártyát

Informális szekvenciadiagram a pénzfelvételre



Hol tartunk?

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

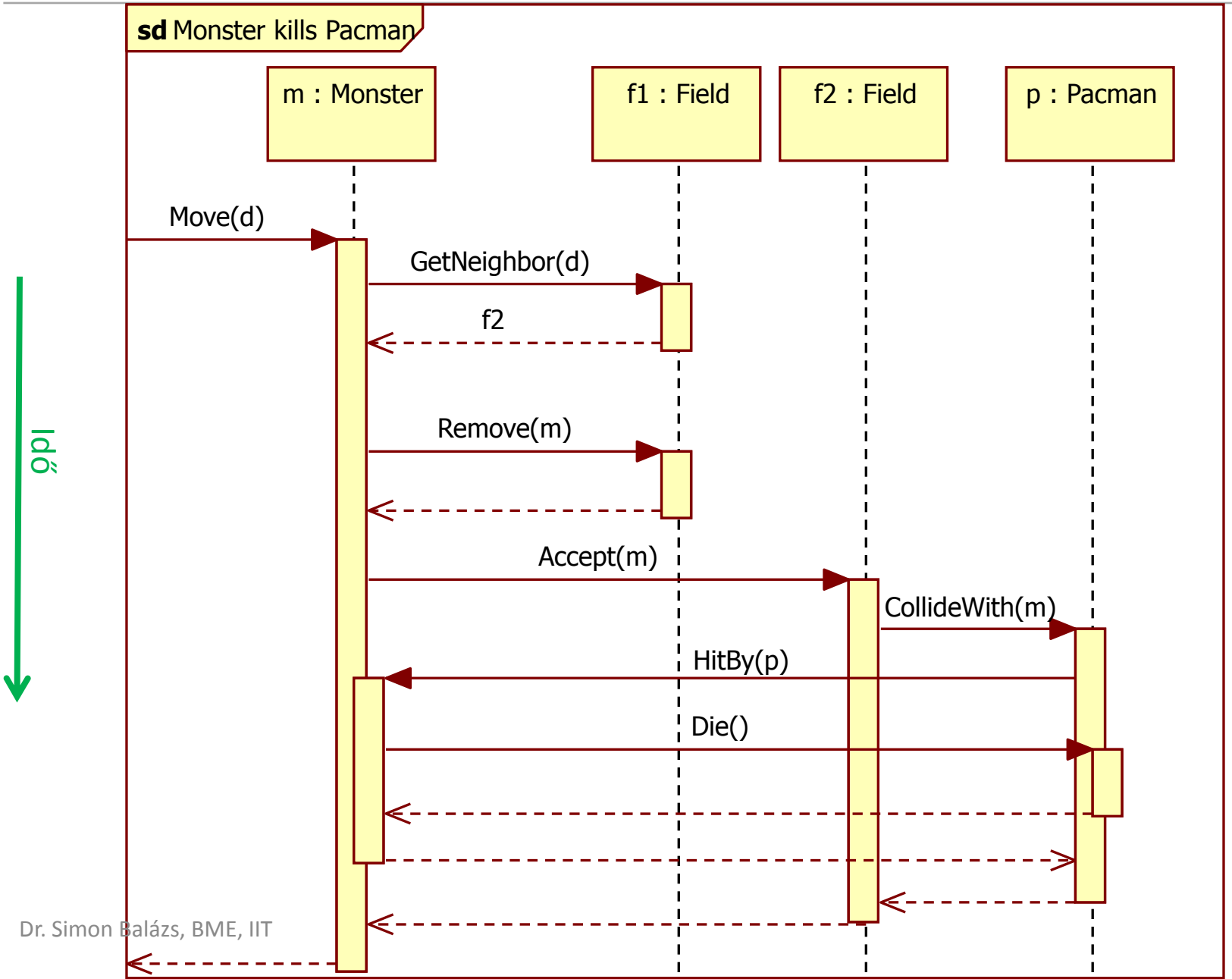
Use case diagram	Aktivitásdiagram	Szekvenciadiagram	Kommunikációs diagram
Állapotdiagram	Időzítődiagram	Interakciós áttekintő diagram	

Kommunikációs diagram (Communication diagram)

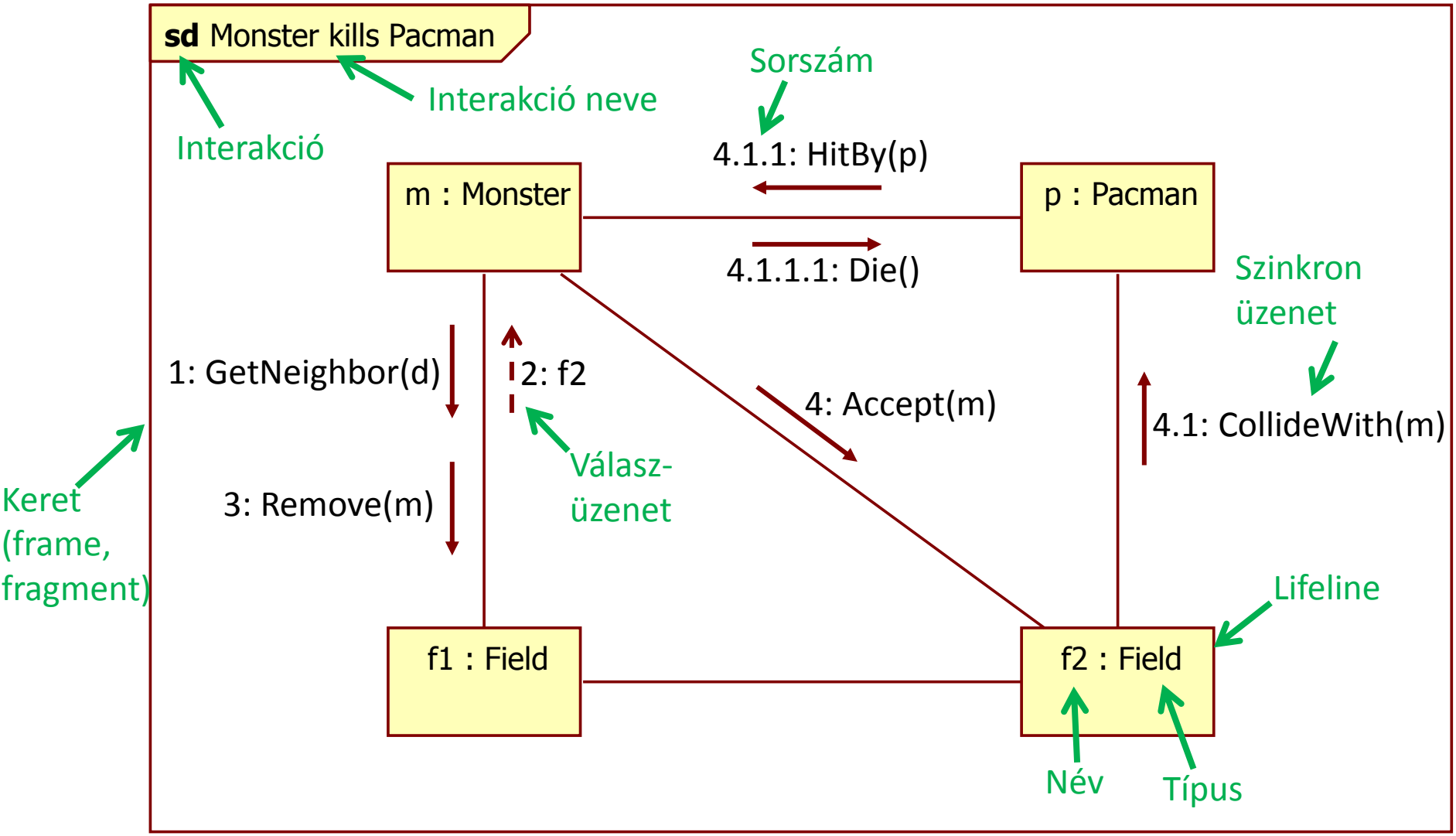
Kommunikációs diagram

- Interakciók grafikus ábrázolására szolgál
  - a rendszer dinamikus viselkedését mutatja
  - az interakciók a részvevők közötti információcserére fókuszálnak
  - az üzenetek sorrendjét hierarchikus számozás határozza meg
- Egy kommunikációs diagram olyan szekvenciadiagramnak felel meg, amelyen nincs strukturális jelölés (interakció használata, combined fragment)
  - a szekvenciadiagramok erőssége a logikai sorrend mutatása, az áttekintő képet azonban nehéz látni
  - a kommunikációs diagramok nagyon jó áttekintő képet adnak (szereplők és kapcsolataik), de nehéz követni a logikai sorrendet
  - a kommunikációs diagram olyan, mintha a szekvenciadiagramot felülről néznénk

Emlékeztető: a szörny megeszi a Pacmant



Kommunikációs diagram: a szörny megeszi a Pacmant



Hol tartunk?

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

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

Interakciós áttekintő diagram (Interaction Overview Diagram)

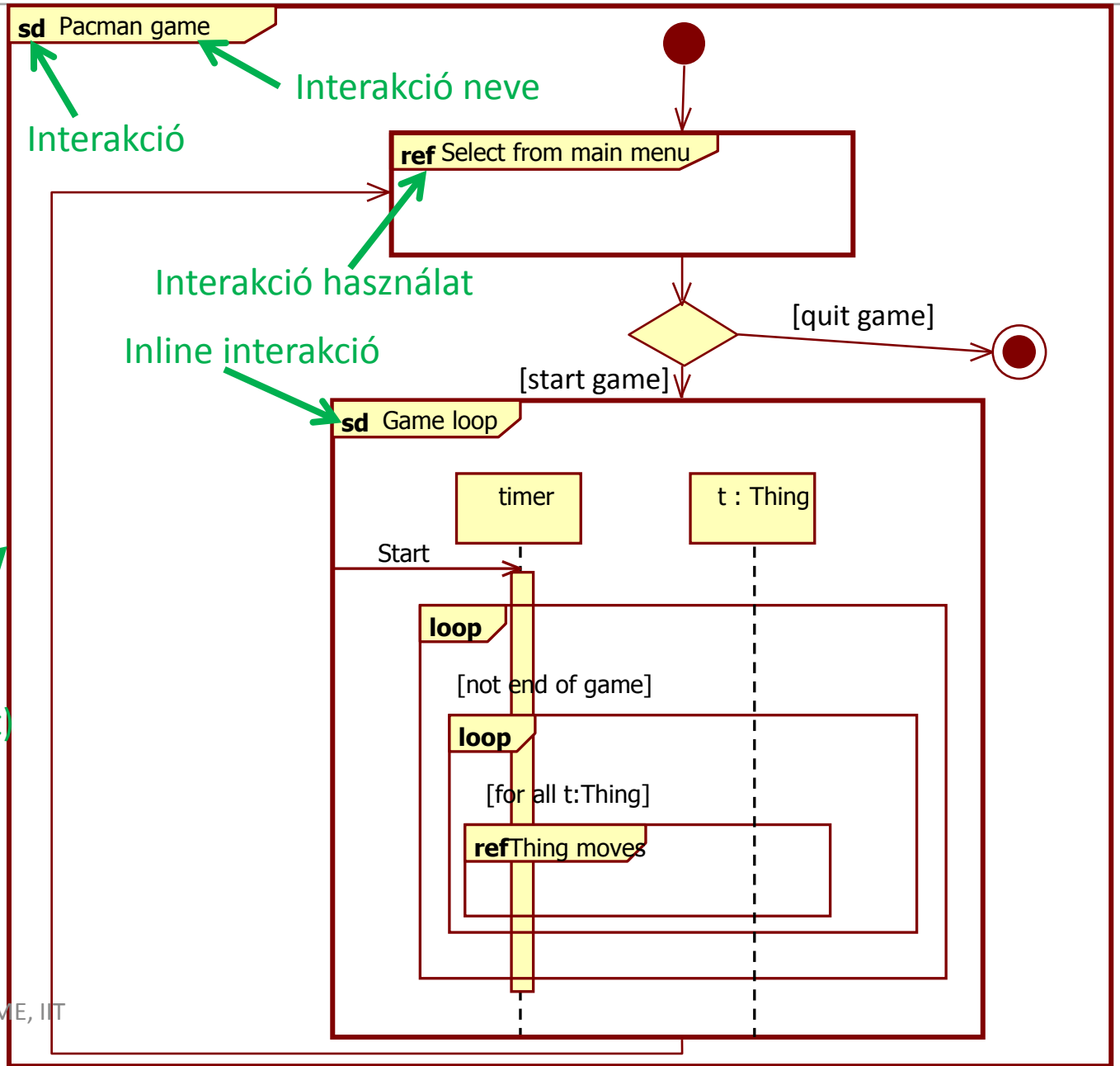
Interakciós áttekintő diagram

- Az interakciós áttekintő diagramok az aktivitásdiagramok egyfajta változatai, amelyek az interakciók egymásutániságáról adnak áttekintő képet
- A különbség az aktivitásdiagramhoz képest: a csomópontok nem akciók, hanem interakciók és interakció használatok
  - Interakciók: inline szekvenciadiagram, kollaborációs diagram, vagy interakciós áttekintő diagram
  - Interakció használatok: referencia egy szekvenciadiagramra, kollaborációs diagramra vagy interakciós áttekintő diagramra

Korlátozások

- További különbségek az aktivitásdiagramokhoz képest:
  - csomópontok: interakciók vagy interakció használatok
  - feltételes combined fragment-ek megfelelője a döntési-merge csomópontpárok
  - a párhuzamos combined fragment-ek megfelelője a fork-join csomópontpárok
  - a ciklus combined fragment-ek megfelelője a gráfban létrehozott körök
  - a feltételes és párhuzamos részeknek hierarchikusan egymásba ágyazottnak kell lennie
  - az interakciós áttekintő diagramok is ugyanolyan interakciós keretben vannak, mint a szekvenciadiagramok és a kollaborációs diagramok

Interakciós áttekintő diagram példa: Pacman játék



Hol tartunk?

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

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

Összefoglalás

- Interakciós grafikus reprezentációja
- A fókusz a részvevők közötti információcserén van
- UML diagramok:
  - **Szekvenciadiagram:** a logikai sorrendet mutatja, az áttekintő képet azonban nehéz látni
  - **Kommunikációs diagram:** nagyon jó áttekintő képet ad (szereplők és kapcsolataik), de nehéz követni a logikai sorrendet
  - **Interakciós áttekintő diagram:** az aktivitásdiagramok egyfajta változata, amely az interakciók egymásutániságáról ad áttekintő képet

Köszönöm a figyelmet!

Unified Modeling Language

Szoftvertechnológia

Dr. Simon Balázs

BME, IIT

Tartalom

- UML diagramok:
  - Állapotdiagram
  - Időzítődiagram
  - Összetett struktúra diagram
  - Profildiagram
- Az UML diagramok összefoglalása
- Az UML-en túl:
  - Object Constraint Language (OCL)
  - XML Metadata Interchange (XMI)
  - MetaObject Facility (MOF)

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

Hol tartunk?

Osztály-diagram ← Egy komponens/rendszer struktúrája objektumorientált modellként

Csomag-diagram ← Csomagok és a közöttük lévő függőségek

Objektum-diagram ← A rendszer részletes állapotának ábrázolása egy adott időpillanatban

Hol tartunk?

Szekvencia-diagram	Az üzenetváltások lehetséges sorozata időben rendezve
Kommunikációs-diagram	Az interakciók áttekintő képe (résztvevők és kapcsolataik)
Interakciós áttekintő diagram	Interakciók ábrázolása az Aktivitásdiagramok egy változataként, amely a vezérlés futásáról ad egy áttekintő képet

Most következik:
Hogyan írjuk le egy komponens/osztály belső állapotát?

Hol tartunk?

Most következik:
Hogyan írjuk le egy komponens/osztály belső állapotát?

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

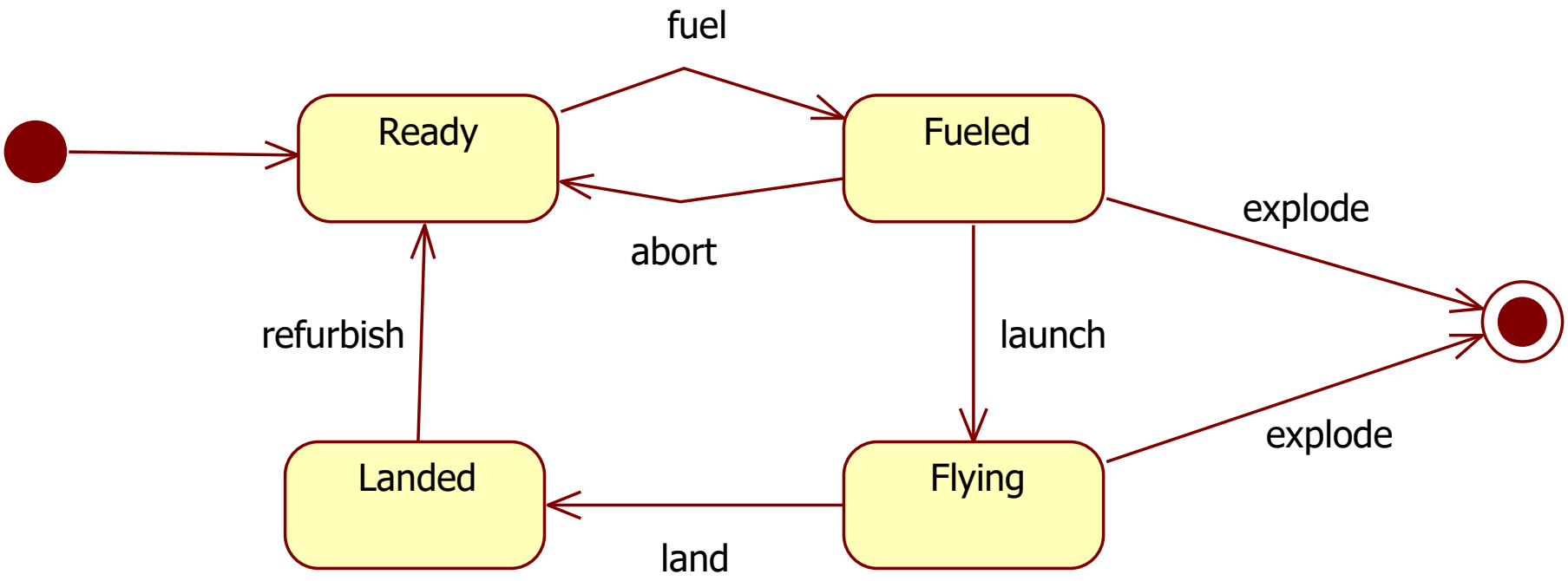
Use case diagram	Aktivitásdiagram	Szekvenciadiagram	Kommunikációs diagram
Állapotdiagram	Időzítődiagram	Interakciós áttekintő diagram	

Állapotdiagram (State Machine Diagram)

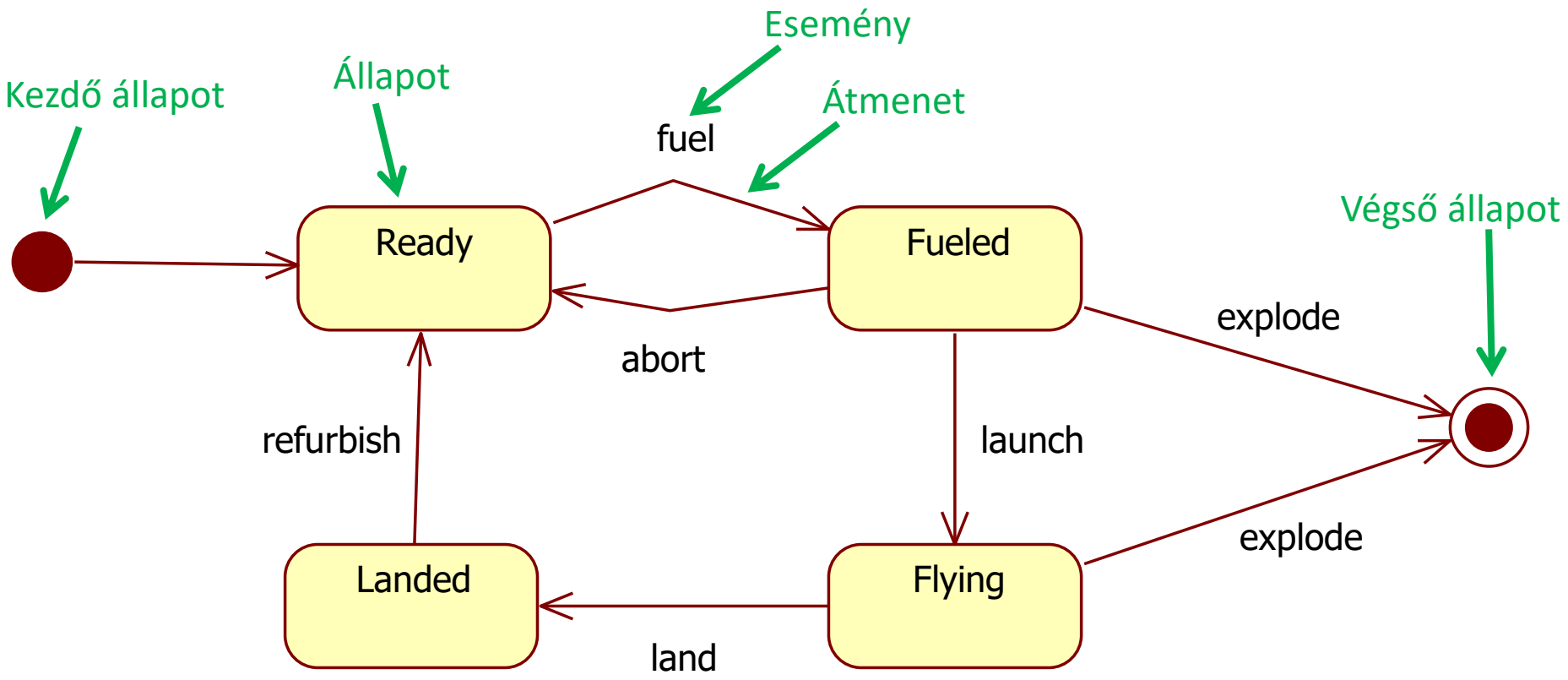
Állapotdiagram

- Diszkrét eseményvezérelt viselkedést ír le véges automaták segítségével formalizálva
- Az állapotok az objektum által tárolt információk különböző kombinációit jelentik
- Az állapotgépek az objektumok lehetséges állapotait ábrázolják, és azt, hogy az objektum hogyan juthat el ezekbe az állapotokba
  - egy objektum állapota akkor változik, ha esemény (függvényhívás) érkezik
- Állapotgépek fajtái:
  - viselkedési állapotgép: egy rendszer részeinek állapotát fejezi ki (pl. osztályok, komponensek viselkedése)
  - protokoll állapotgép: érvényes interakciók sorozatát fejezi ki
- (A következőkben: viselkedési állapotgép. A protokoll állapotgépet ld. a szabványban.)

Állapotdiagram példa: újrahasznosítható rakéta



Állapotdiagram példa: újrahasznosítható rakéta

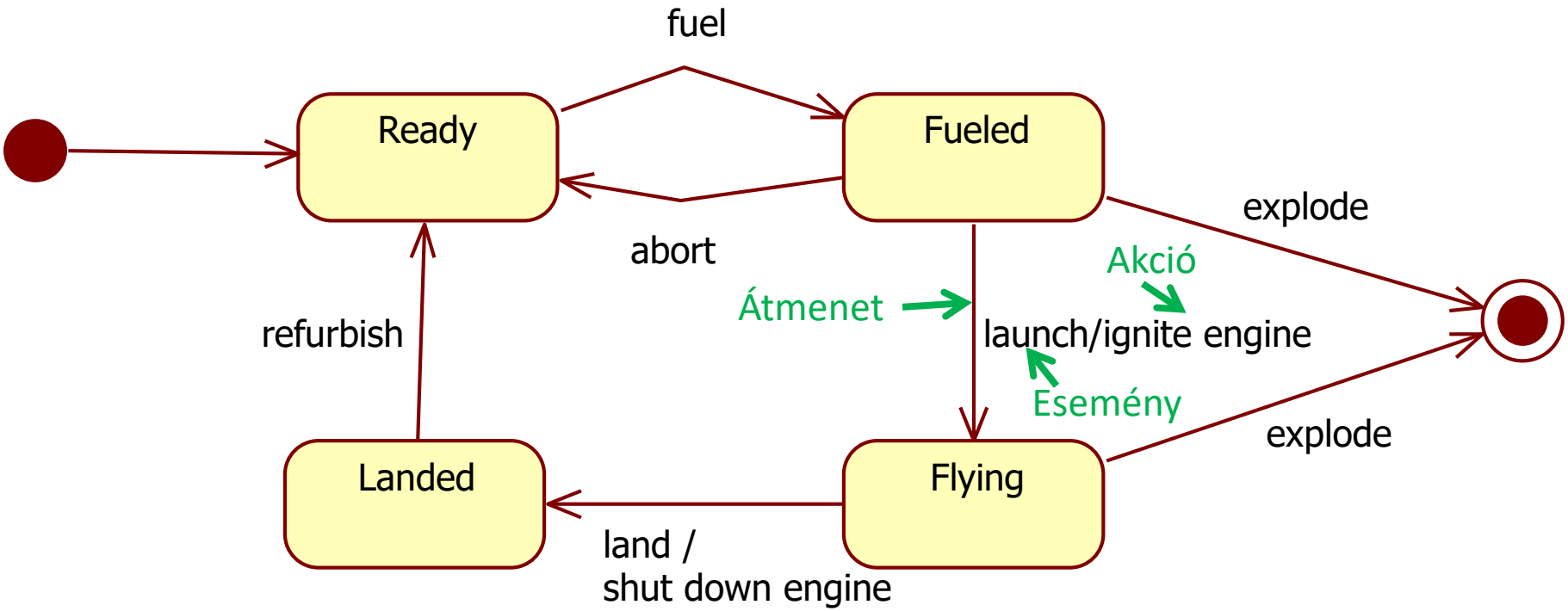


Állapotdiagram

- **Kezdő állapot (initial state)**:
  - a futás a kezdőállapotból kiinduló átmenettel indul
- **Végső állapot (final state)**:
  - speciális állapot, az objektum futásának végét jelzi
- **Állapot (state)**:
  - a futás egy olyan helyzetét reprezentálja, amikor egy invariáns feltétel teljesül
    - tehát az objektum attribútumainak értékei valamilyen feltételt teljesítenek
  - nem feltétlenül teljesen statikus helyzetet jelent
    - az állapotot meghatározó attribútumok értékei változhatnak, feltéve, hogy továbbra is teljesül az invariáns feltétel
- **Átmenet (transition)**:
  - egy lehetséges mozgást ábrázol a forrásállapotból a célállapotba, ha a meghatározott esemény bekövetkezik
  - a nyíl mellé írt címke mutatja az eseményt
- **Esemény (event)**:
  - ha az esemény bekövetkezik, az aktuális állapot átvált annak az átmenetnek célállapotára, amelynek forrásállapota az aktuális állapot, eseménye pedig a bekövetkezett esemény
  - ha több lehetséges átmenet is van, csak egy állapotváltás történik nem-determinisztikusan
  - ha nincs lehetséges átmenet, nem történik állapotváltás
  - egy esemény tipikusan az objektumon hívott metódus (üzenet)

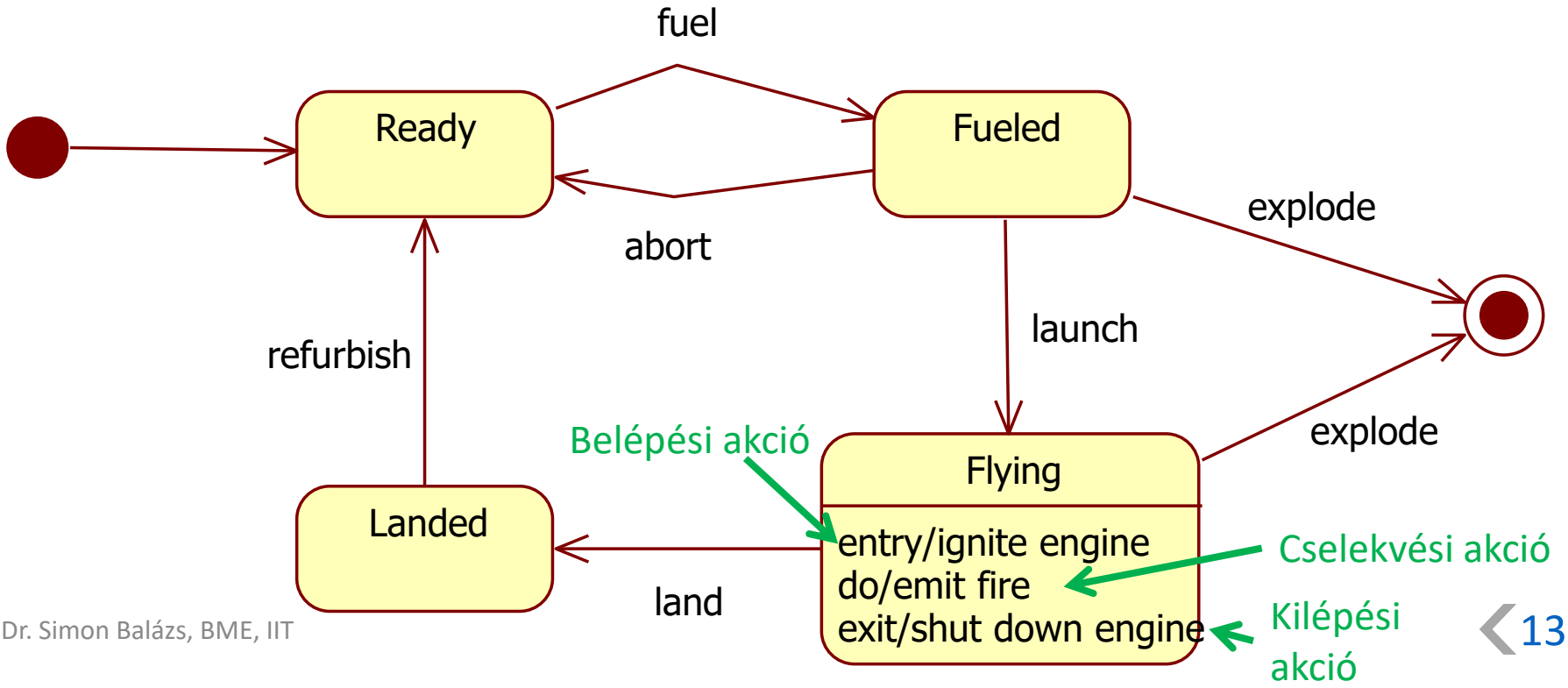
Állapotdiagram: Akciók (actions)

- Az állapotgép az eseményekre akciók végrehajtásával reagálhat
  - pl. egy változó értékének megváltoztatása, I/O művelet, függvény meghívása, másik esemény generálása



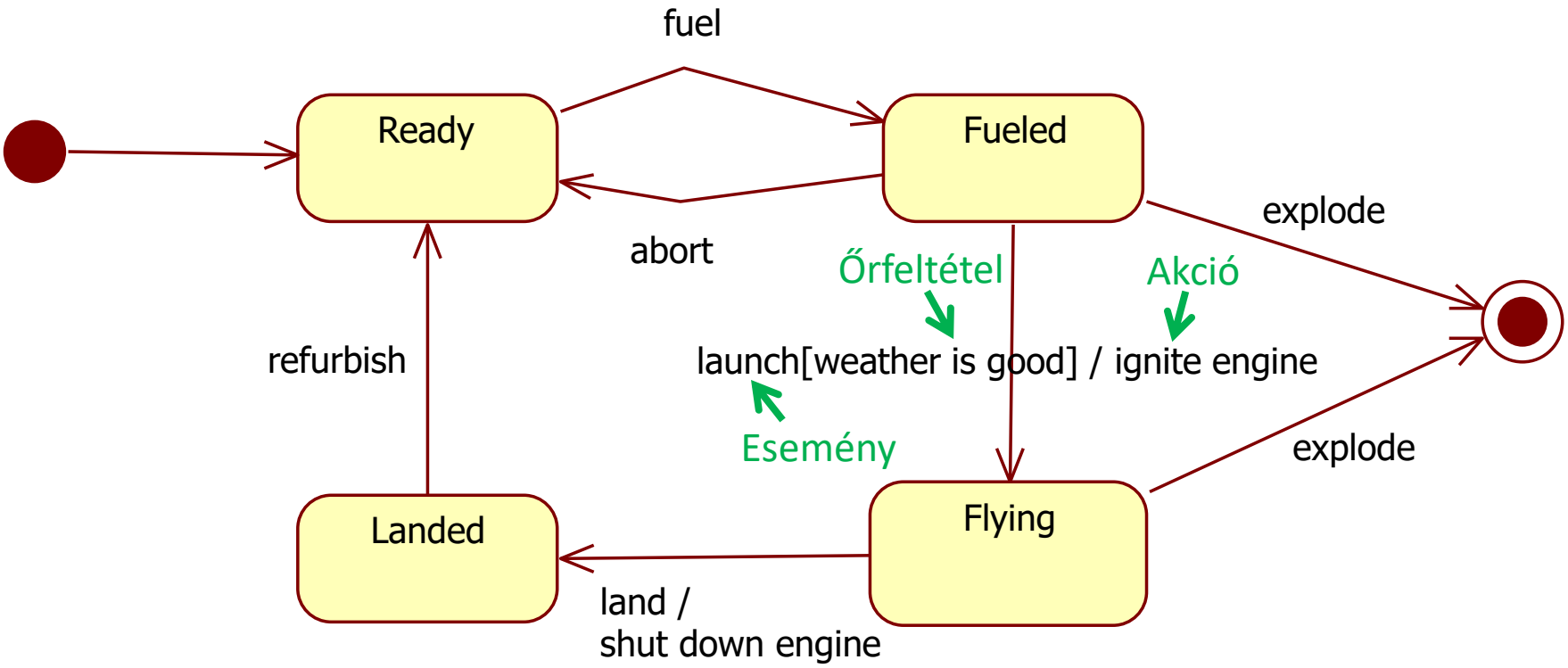
Állapotdiagram: Akciók (actions)

- Állapoton belül is lehet akció:
  - **Belépési akció (entry action)**: akkor fut le, ha az állapotba egy külső állapotátmenettel lépünk be
  - **Kilépési akció (exit action)**: akkor fut le, ha kilépünk az állapotból
  - **Cselekvési akció (do action)**: a végrehajtása akkor kezdődik, miután beléptünk az állapotba (a belépési akció után), és mindaddig fut, amíg véget nem ér, vagy ki nem lépünk az állapotból
- Egy állapotnak több belépési-, kilépési- és cselekvési akciója is lehet



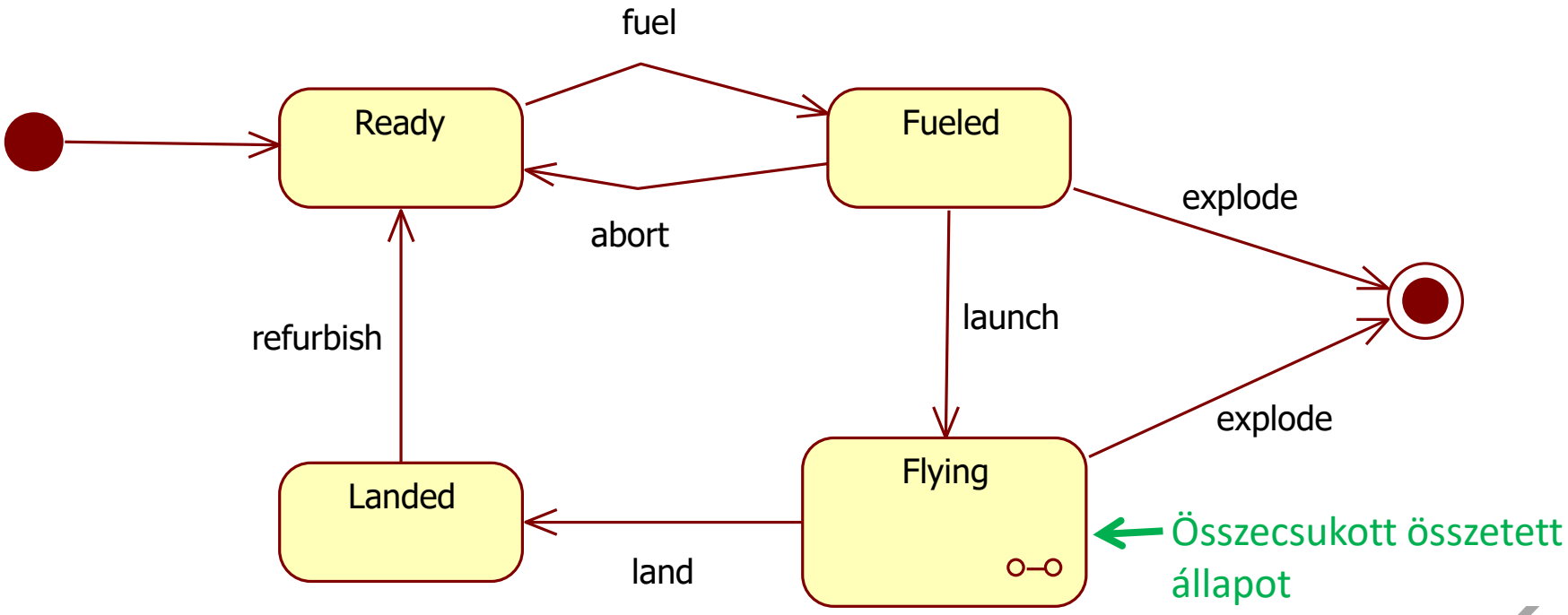
Őrfeltétel (guard condition)

- Az átmenet csak akkor aktív, ha a hozzárendelt őrfeltétel értéke igaz
- Ha egy átmenet nem aktív, akkor nem fut le, még akkor sem, ha a megfelelő esemény érkezik



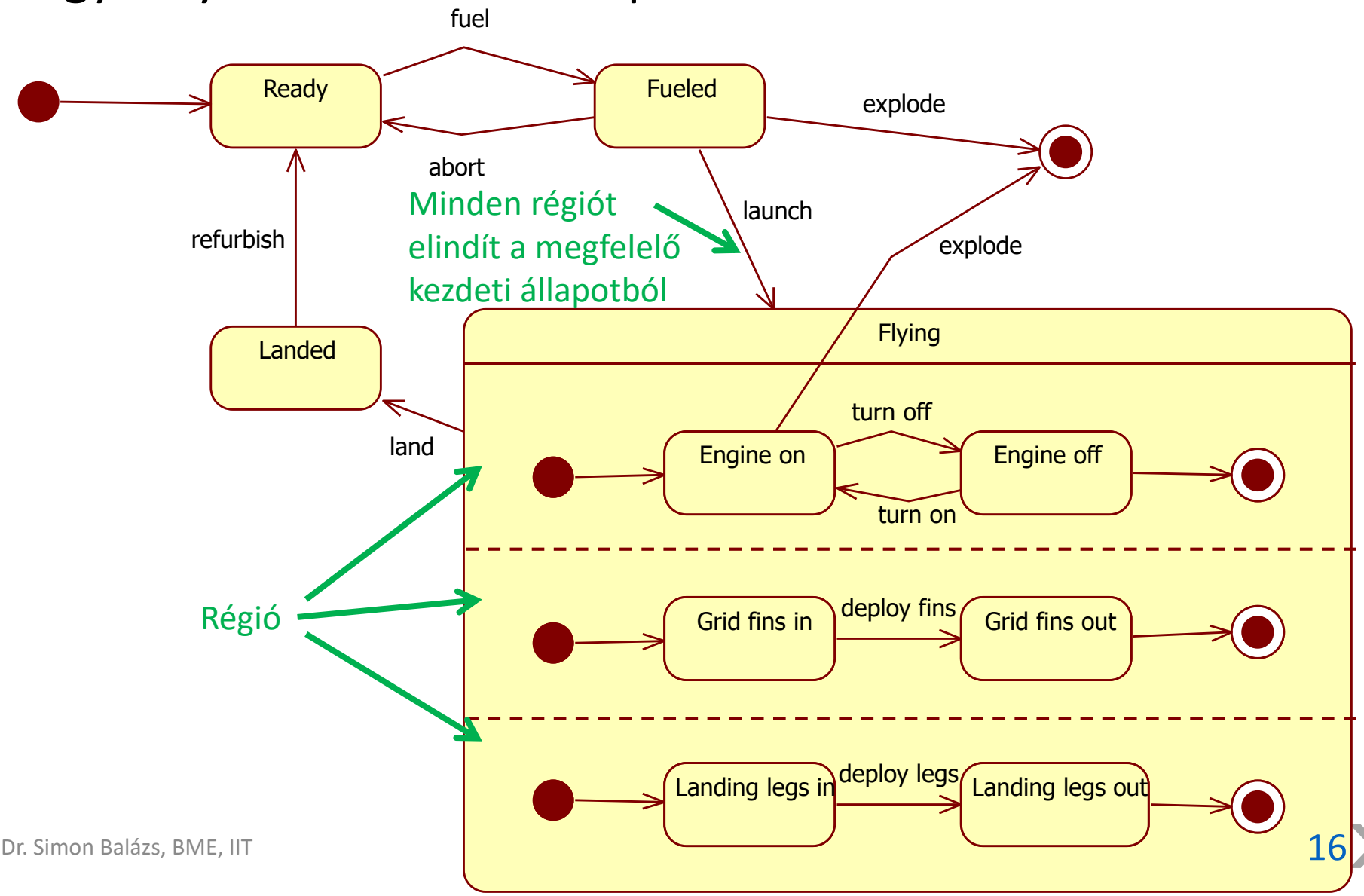
Összetett állapot (composite state)

- Egy egyszerű állapotnak nincsenek belső állapotai vagy átmenetei
- Egy összetett állapotnak van belső struktúrája
  - egy vagy több régió (region)
  - minden egyes régióban állapotok és átmenetek
  - a régiók függetlenek egymástól
- Az olvashatóság kedvéért egy összetett állapot összecsukható egyetlen állapottá



Összetett állapot (composite state)

- Egy kinyitott összetett állapot:

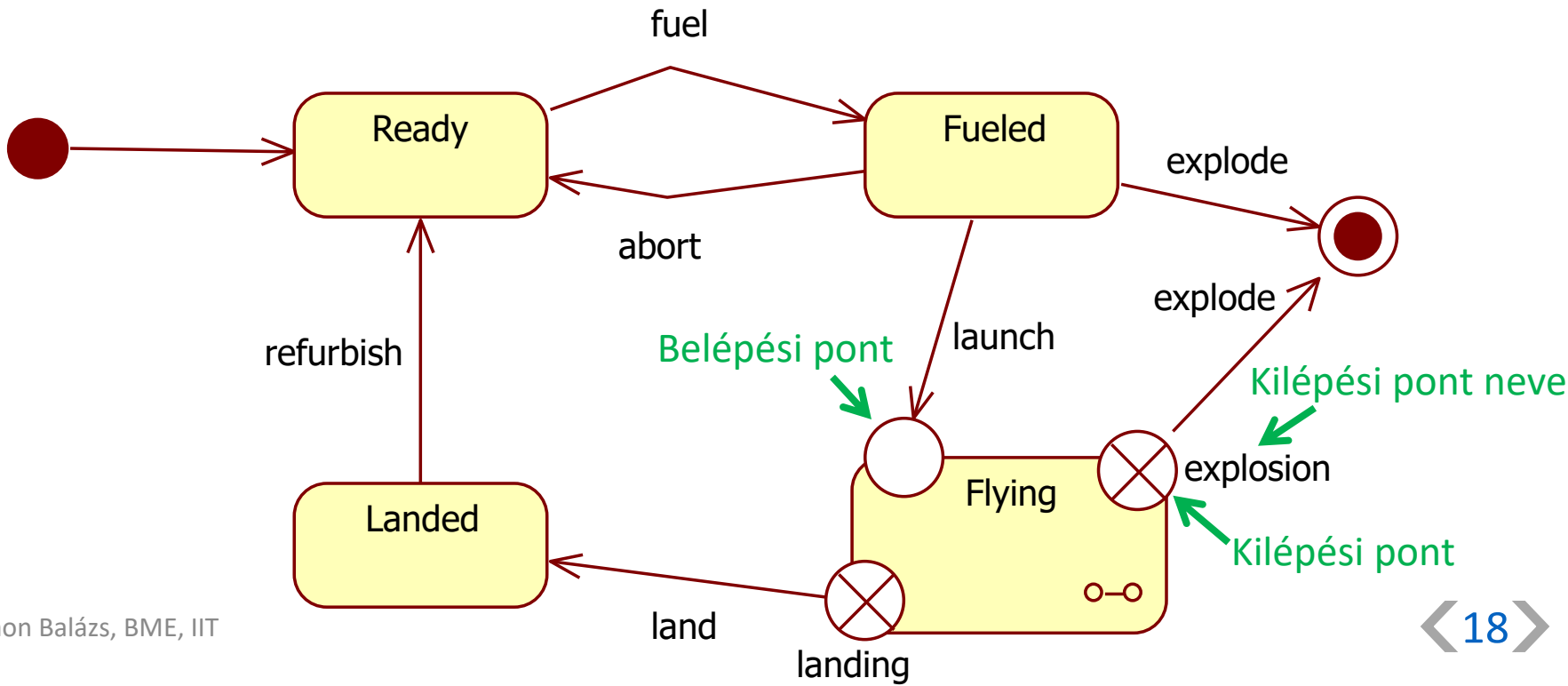


Belépés és kilépés összetett állapotokban

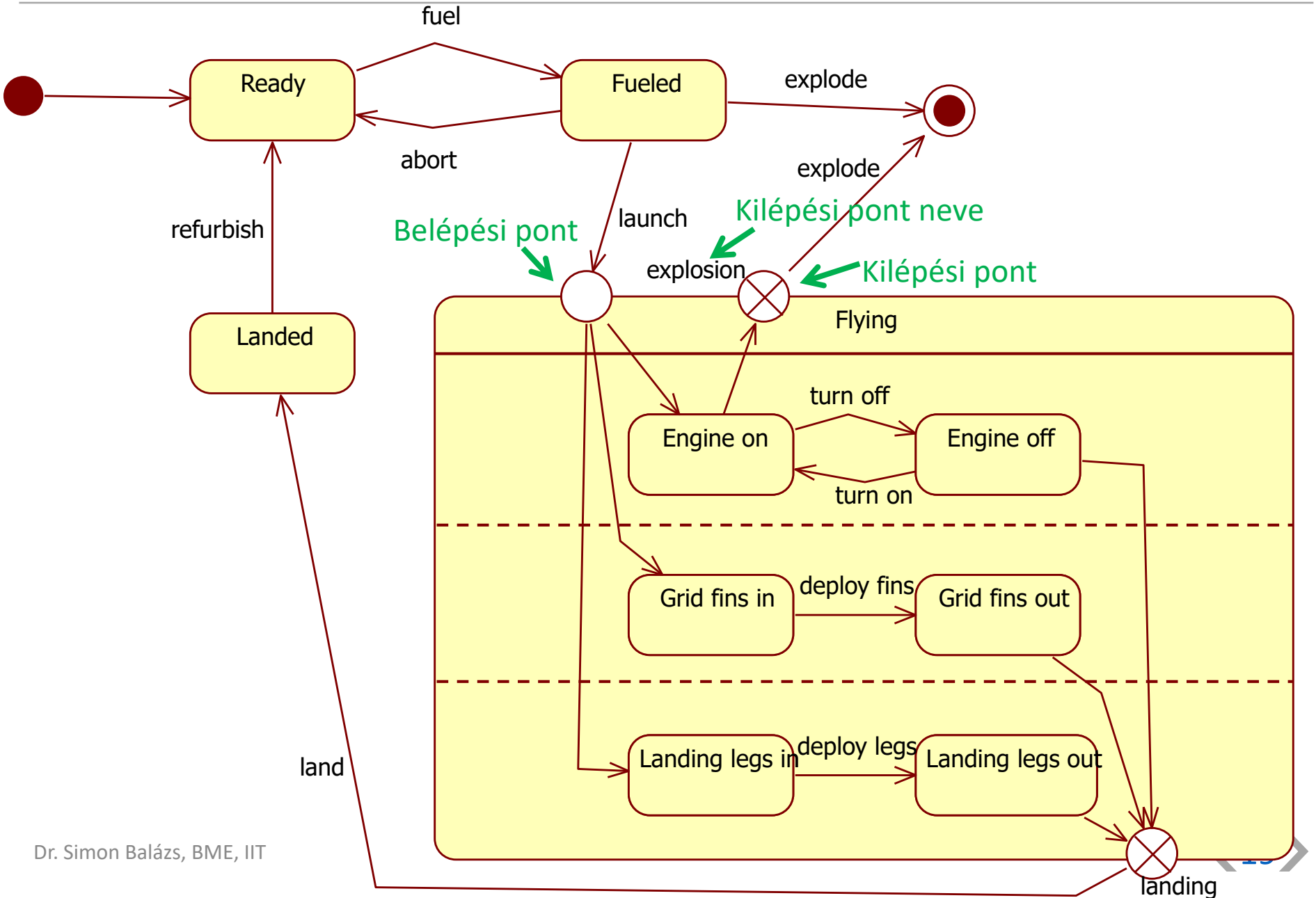
- **Alapértelmezett aktiválás (default activation)**: az átmenet, amely közvetlenül az összetett állapot peremébe megy, minden régiót párhuzamosan elindít a megfelelő kezdeti állapotokból
  - ha nincs kezdeti állapota egy régiónak, a viselkedés nem definiált
- **Explicit aktiválás (explicit activation)**: az átmenet, amely egy összetett állapot belső állapotába fut be, az adott régiót ebből az állapotból indítja, a többi régió alapértelmezett aktiválással indul
- Az összetett állapotból kimenő címkézetlen átmenet csak akkor fut le, ha minden régió eljutott a saját végső állapotába

Belépési és kilépési pontok (entry and exit points)

- A belépési és kilépési pontok az összetett állapot egységbezárását segítik
  - bizonyos helyzetekben hasznos lehet elrejteni egy összetett állapot belsejét, és nem engedni közvetlen átmenetet belülre
  - a belépési és kilépési pontok segítenek a külső átmeneteket összekötni a belső elemekkel



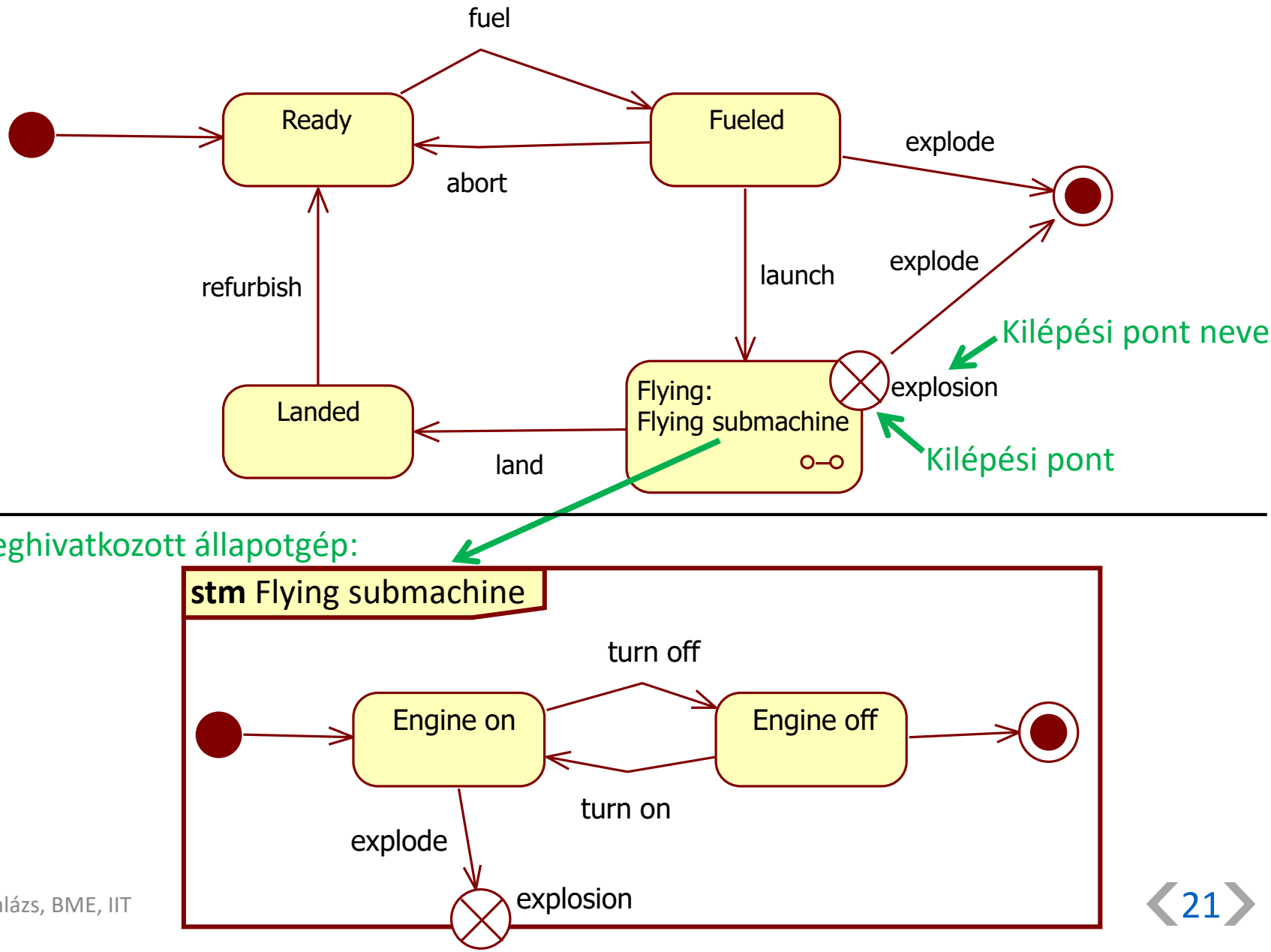
Belépési és kilépési pontok (entry and exit points)



Alállapotgép állapot (submachine state)

- Alállapotgép állapotok segítségével egy állapotgép leírása többször újrahasznosíthatóvá válik
- Hasonló az összetett állapothoz, azonban az alállapotgép állapotok külön-külön példányokat jelentenek a hivatkozott állapotgépből
  - az alállapotgép állapot olyan, mint egy C makró meghívása: mintha bemásolnánk oda a hivatkozott állapotgépet
- A bemenő és kimenő állapotokat az alállapotgép állapothoz kell kötni, de ezek függnek attól a kontextustól, ahol az alállapotgép állapotot használjuk

Alállapotgép állapot (submachine state) példa

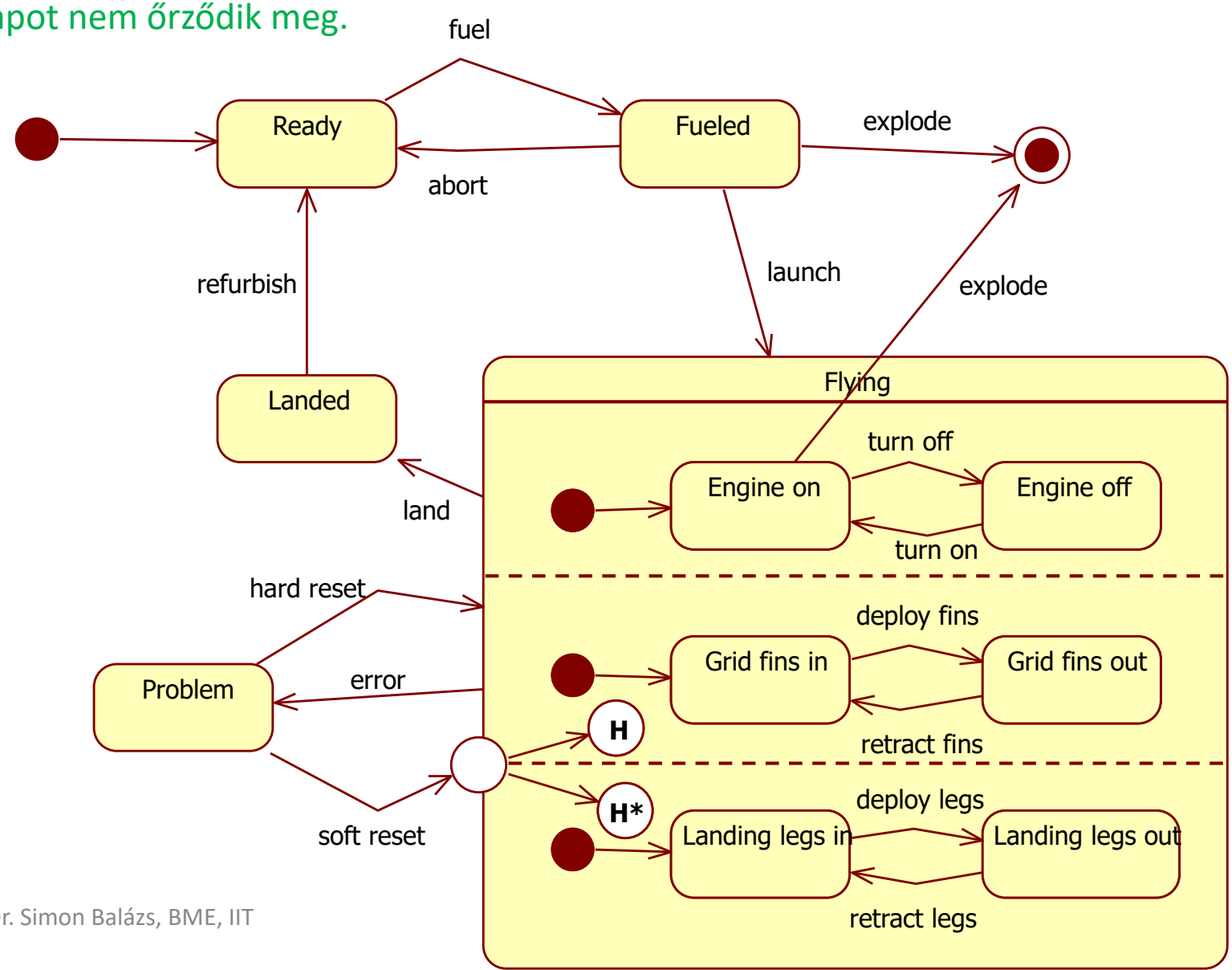


Történet (history)

- A történet állapotok (history states) egy összetett állapot régiói korábbi konfigurációjának visszaállítását segítik, amely akkor volt érvényes, amikor utoljára kiléptünk az összetett állapotból
- Az állapot (konfiguráció) akkor áll vissza, ha az aktív átmenet a történet állapotban ér véget
- Ha nincs korábbi állapotkonfiguráció (vagyis először lépünk be egy összetett állapotba, és ezt egy történet állapoton keresztül tesszük):
  - ha a történet állapotnak van átmenete egy alállapotba, akkor a történet állapot kezdeti állapotként viselkedik
  - egyébként az alapértelmezett aktiválás érvényes
- Kétfajta történet állapot van:
  - **mély történet (deep history)**: visszaállítja az összetett állapot konfigurációját, és minden alállapot konfigurációját rekurzív módon
  - **sekély történet (shallow history)**: csak a legkülső összetett állapot konfigurációját állítja vissza, az alállapotokét nem

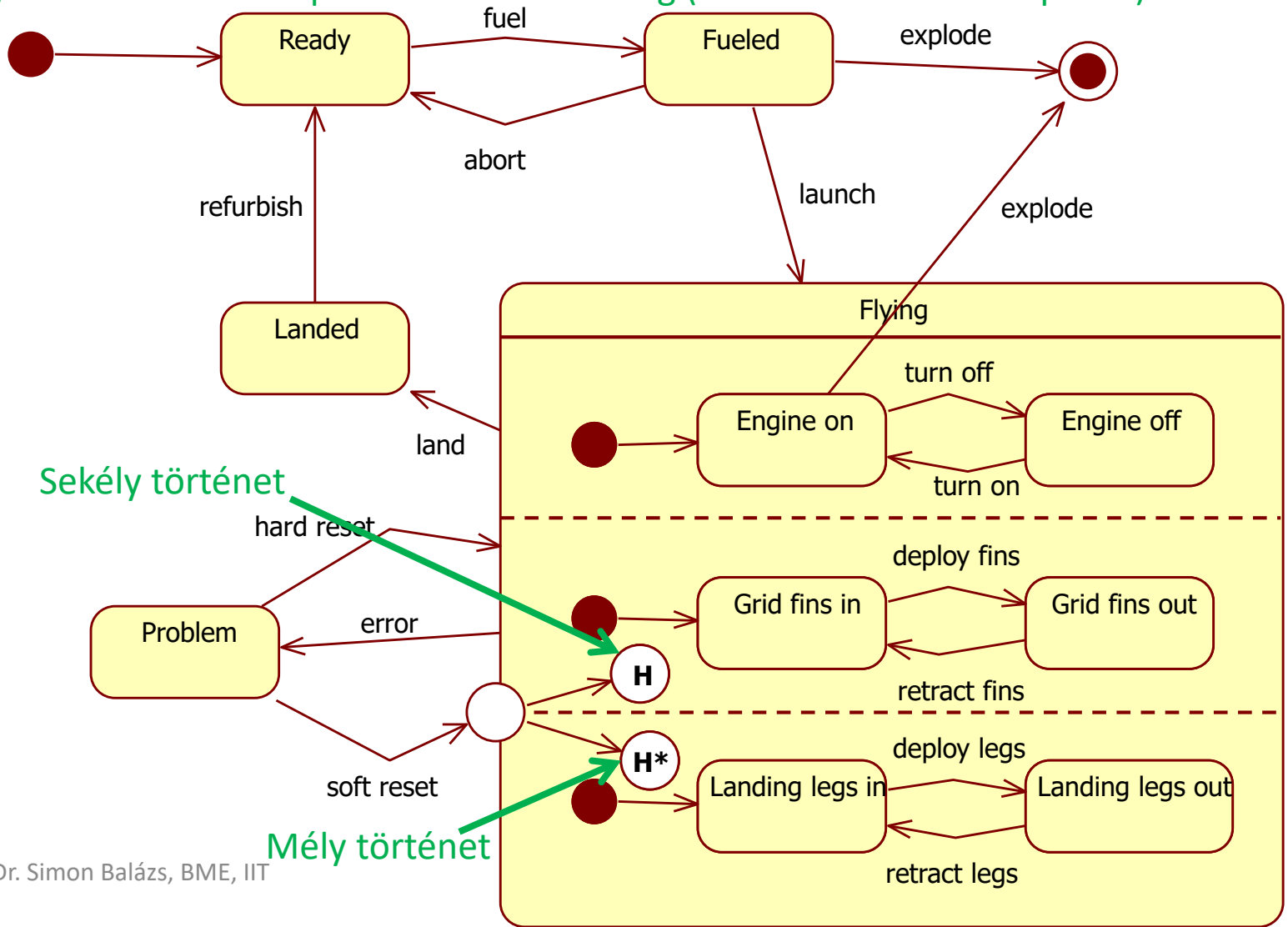
Történet (history) példa

‘hard reset’ esetén: the engine is turned on, grid fins are pulled in, landing legs are pulled in. Állapot nem őrződik meg.



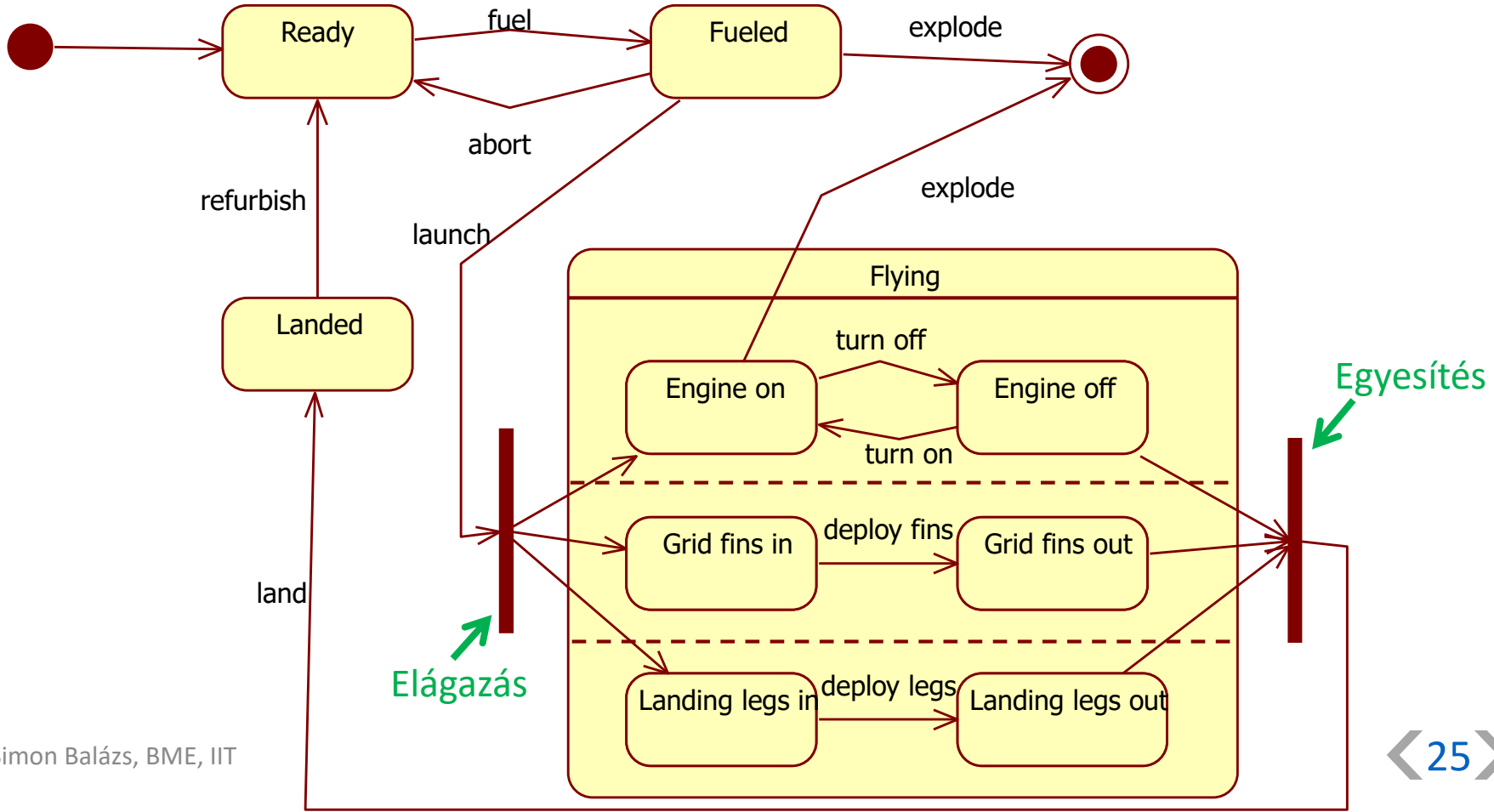
Történet (history) példa

‘soft reset’ esetén: the engine is turned on, grid fins and landing legs preserve their positions.
A lábak belső állapota megőrződik (ha ezek összetett állapotok).
A grid fin-ek belső állapota **nem** őrződik meg (ha ezek összetett állapotok).

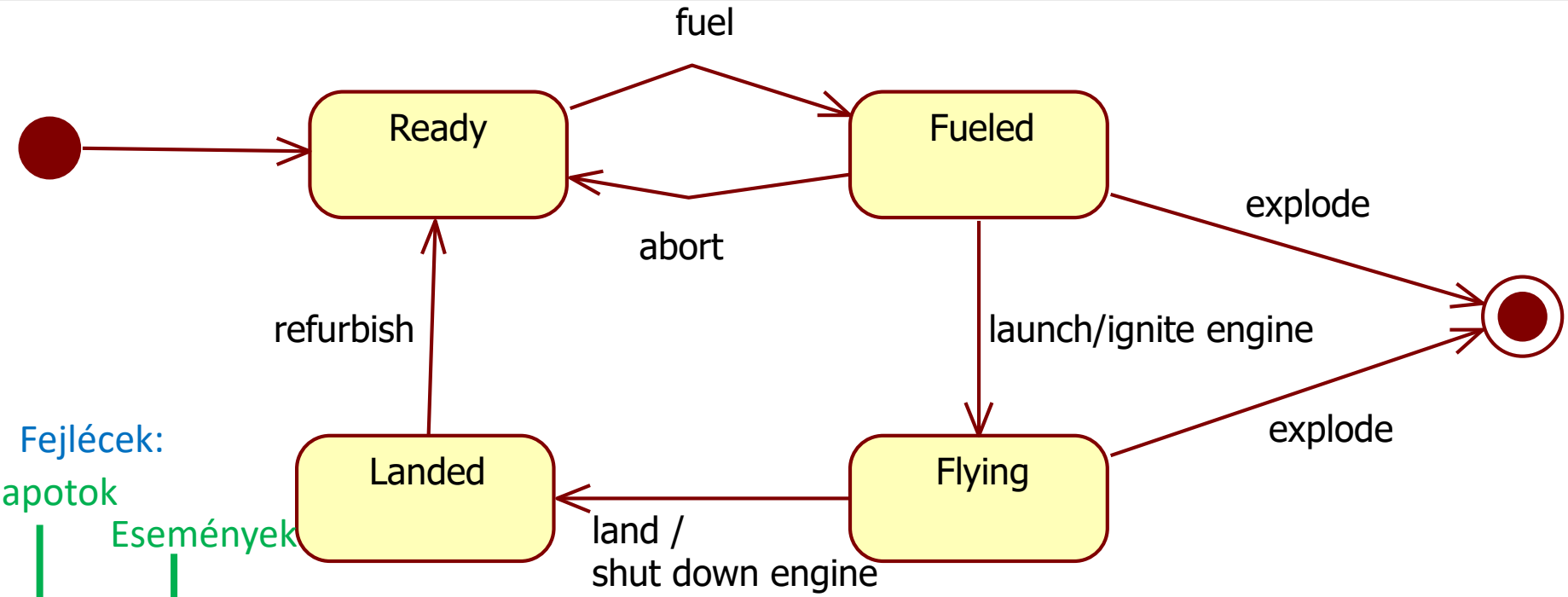


Elágazás és egyesítés (fork and join)

- **Elágazás (fork)**: a beérkező átmenet két vagy több átmenetre bontása, amelyek egy összetett állapot különböző régióiba futnak be
  - a kimenő átmeneteknek nem lehet őrfeltétele vagy eseménye
- **Egyesítés (join)**: több átmenet összevárása, majd egy átmenetként folytatás
  - a bejövő átmeneteknek nem lehet őrfeltétele vagy eseménye



Állapotgép táblázatos formában

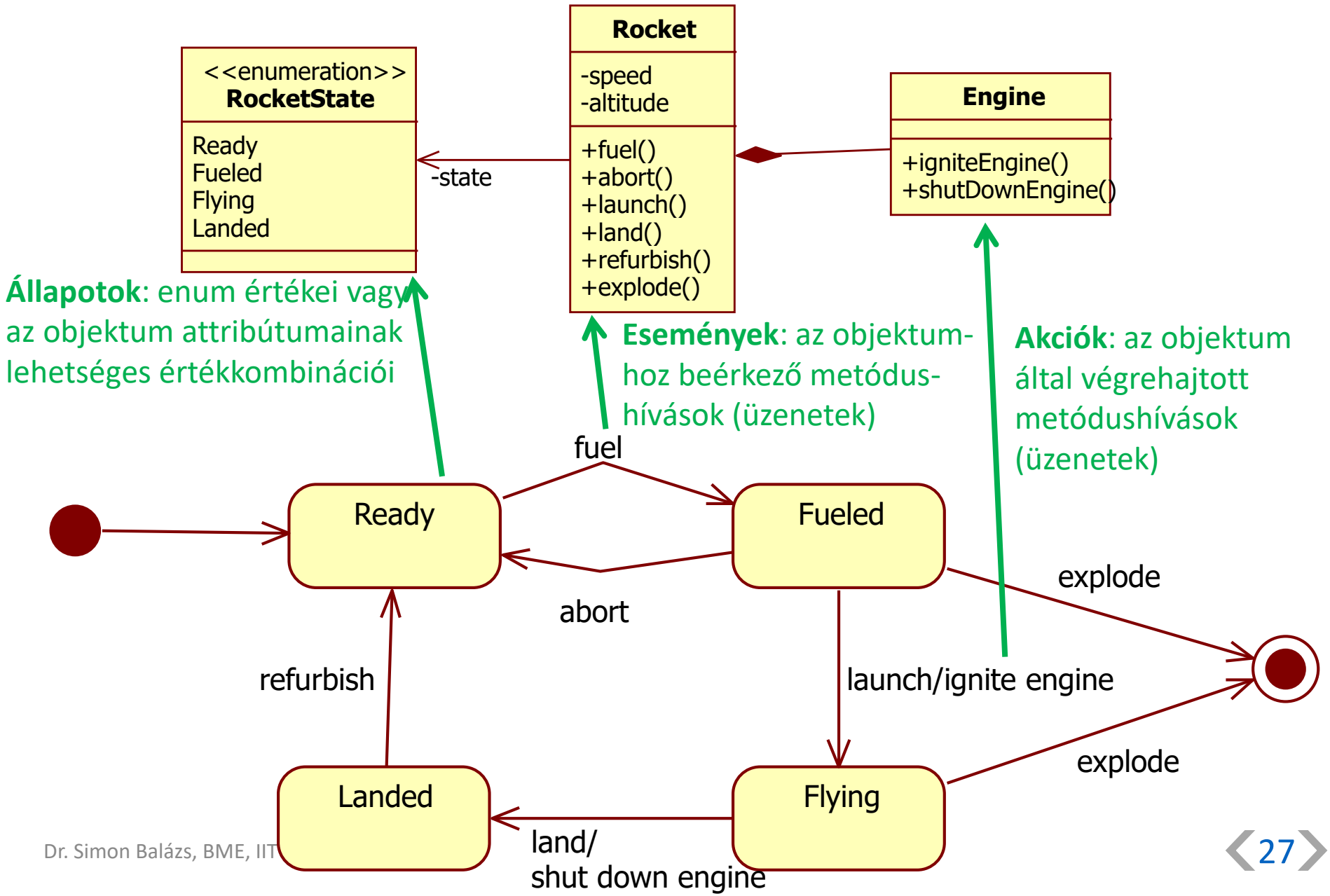


Fejlécek:
Állapotok
Események

	fuel	**abort**	**launch**	**land**	**refurbish**	**explode**
Ready	Fueled/-	-/-	-/-	-/-	-/-	-/-
Fueled	-/-	Ready/-	Flying/ ignite engine	-/-	-/-	-/-
Flying	-/-	-/-	-/-	Landed/ shut down engine	-/-	-/-
Landed	-/-	-/-	-/-	-/-	Ready/-	-/-

Cellák: Következő állapot / Akció

Állapotgép a Rocket osztályra



Hol tartunk?

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

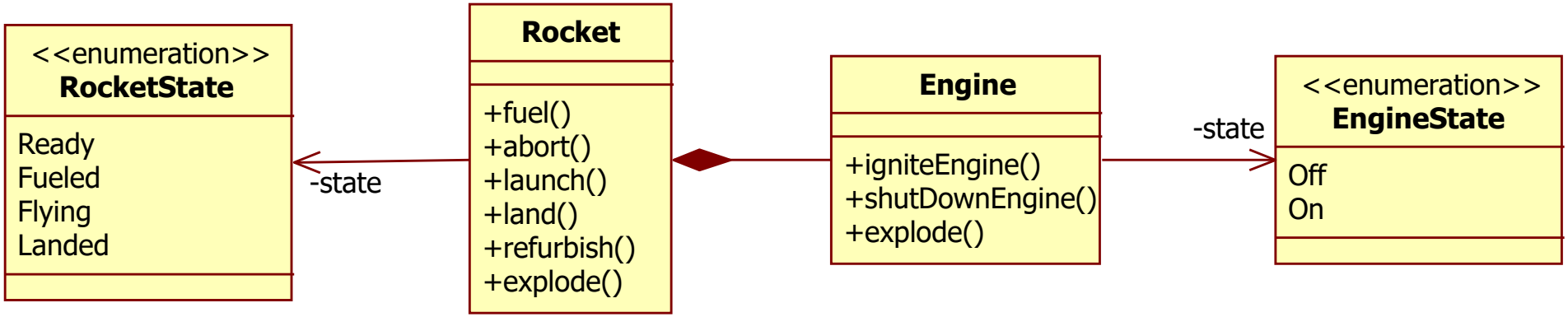
Use case diagram	Aktivitásdiagram	Szekvenciadiagram	Kommunikációs diagram
Állapotdiagram	Időzítődiagram	Interakciós áttekintő diagram	

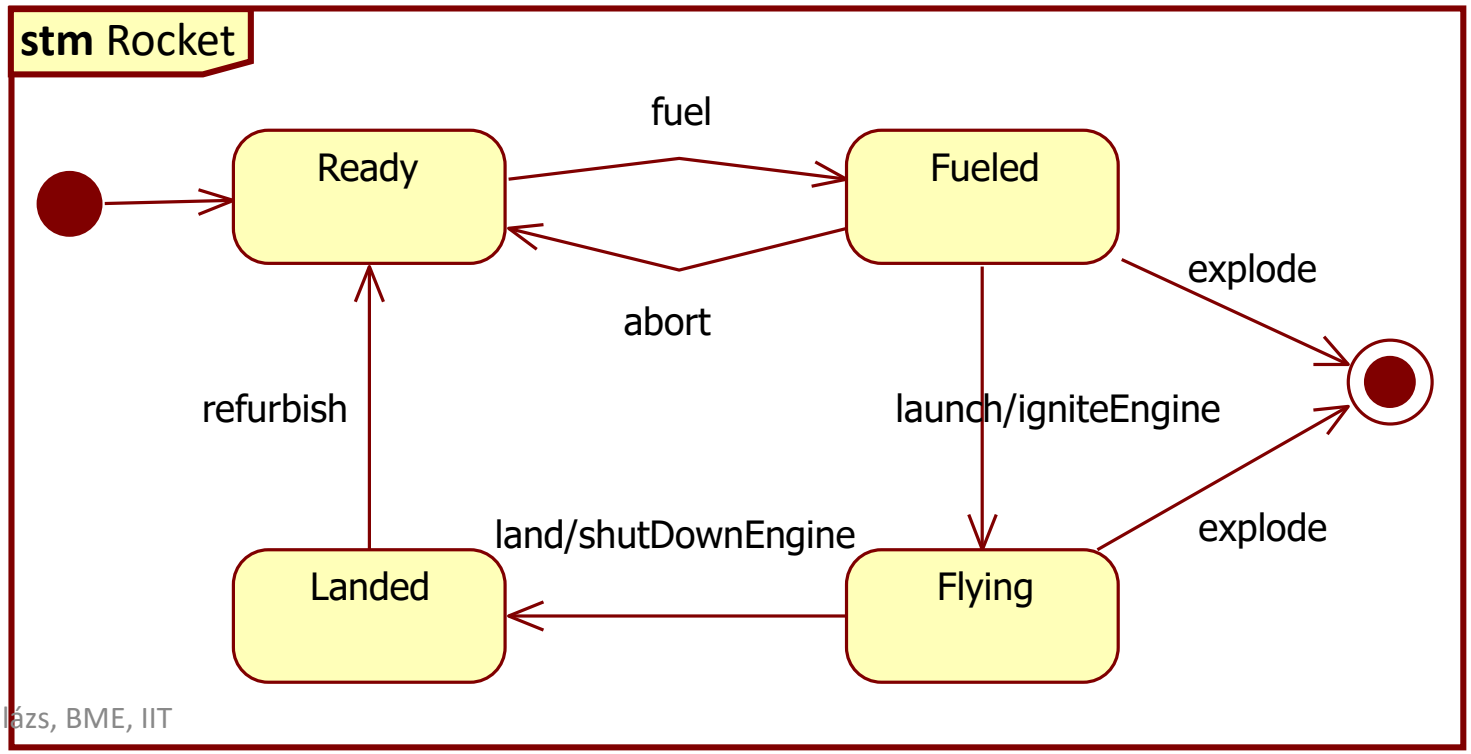
Időzítődiagram (Timing Diagram)

Időzítődiagram (Timing Diagram)

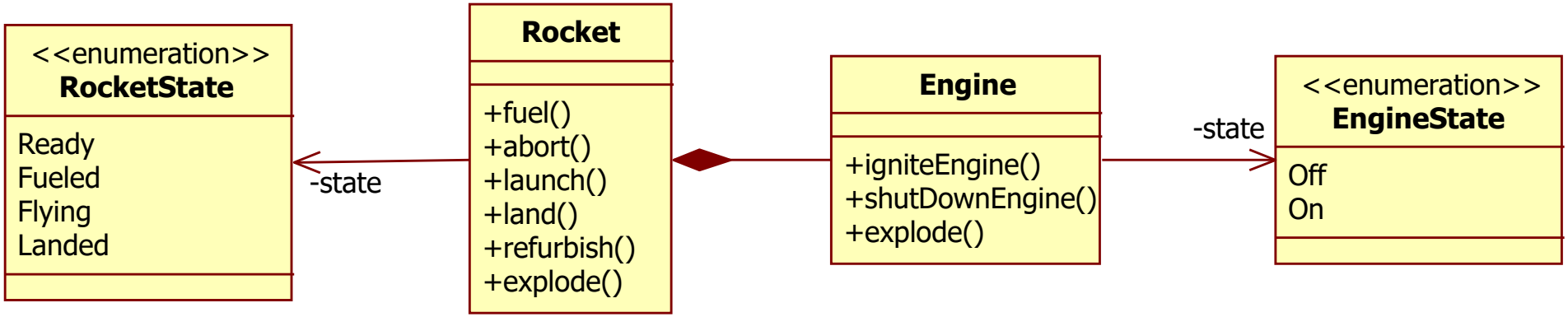
- Az időzítődiagram lifeline-okon belüli és azok közötti állapotváltásokra fókuszálnak egy időtengely mentén
- Az időzítődiagramok önálló classifier-eket és classifier-ek közötti interakciókat ábrázolnak eseményekkel és állapotváltásokkal, követve az időt és az ok-okozati összefüggéseket
- Az időzítődiagram hasonló a szekvenciadiagramhoz, de:
  - az idő balról jobbra telik (nem fentről lefelé)
  - a lifeline-ok állapotát is mutatja
    - az állapot lehet diszkrét vagy folytonos

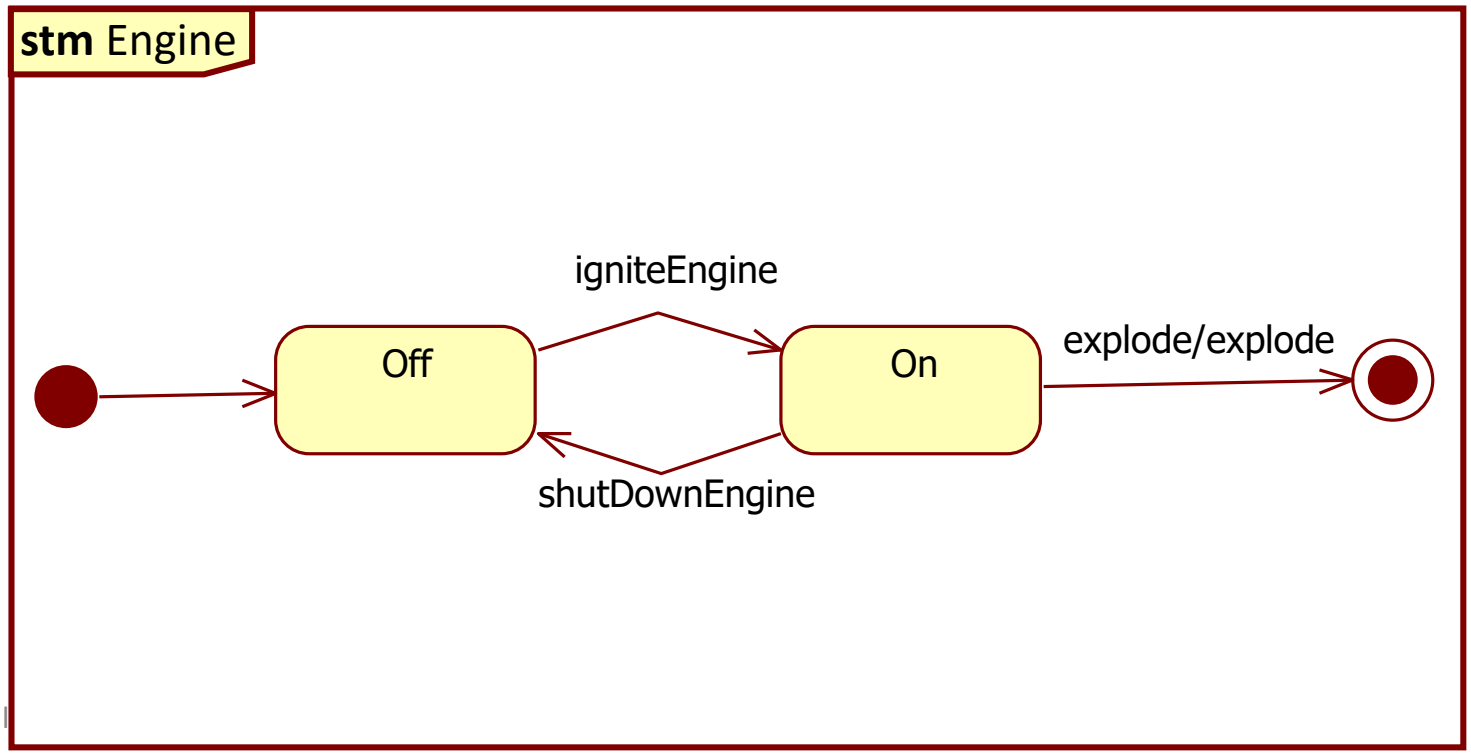
Állapotdiagram a Rocket osztályra



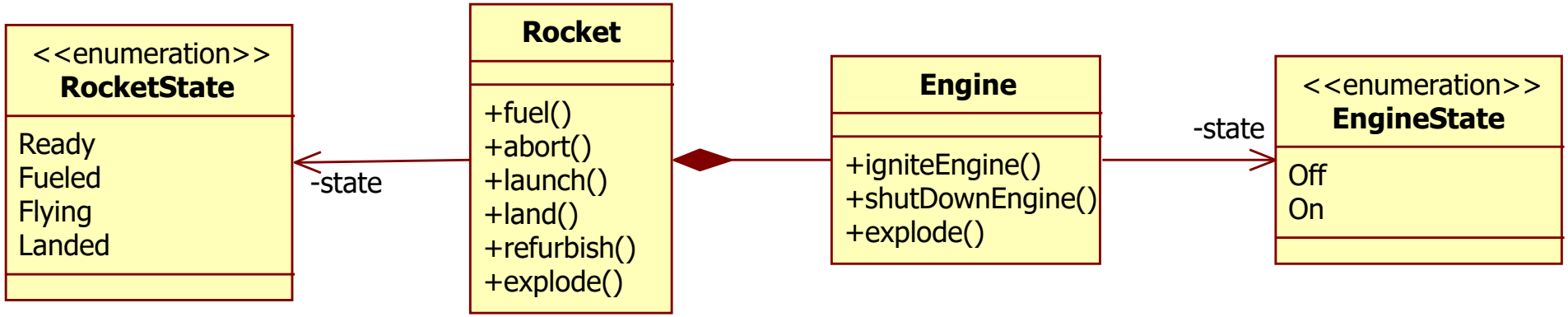


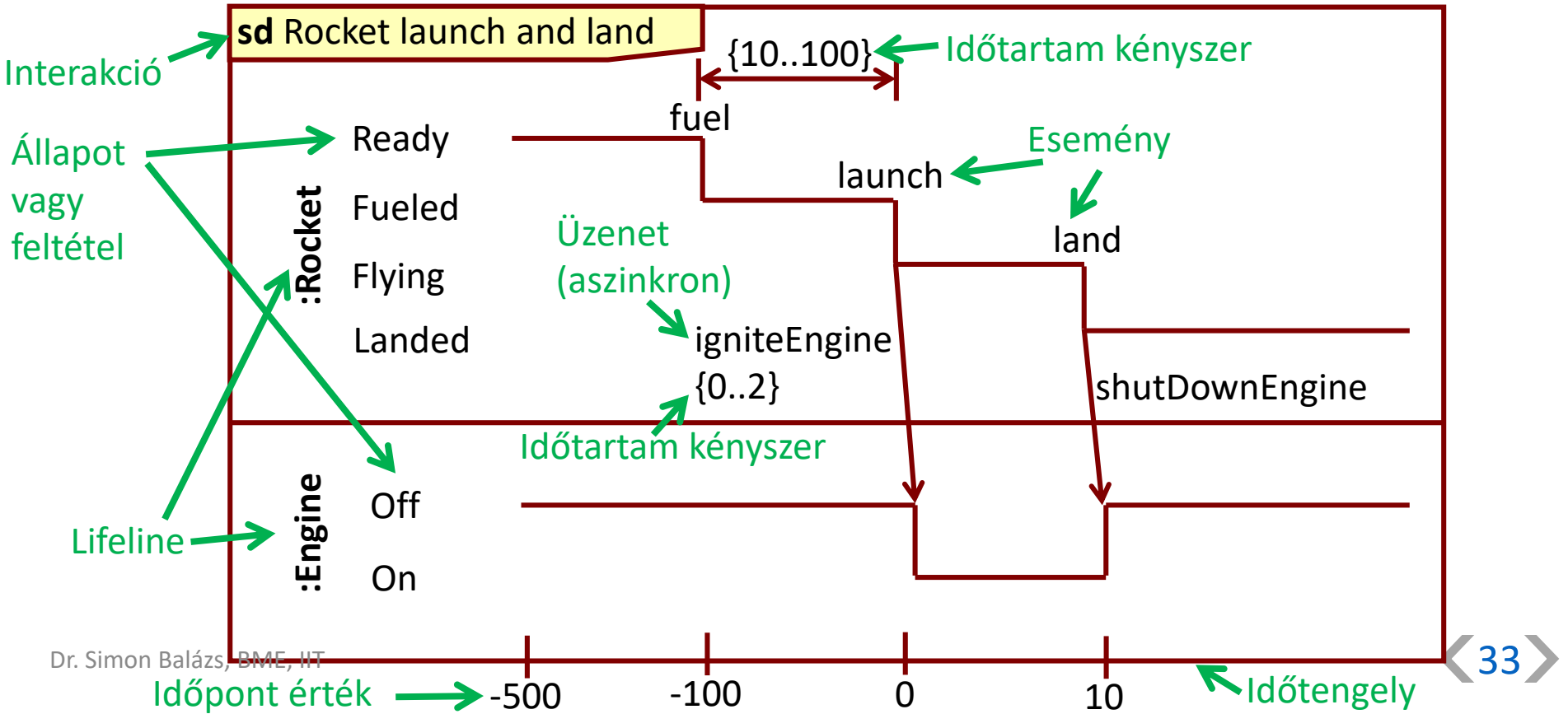
Állapotdiagram az Engine osztályra



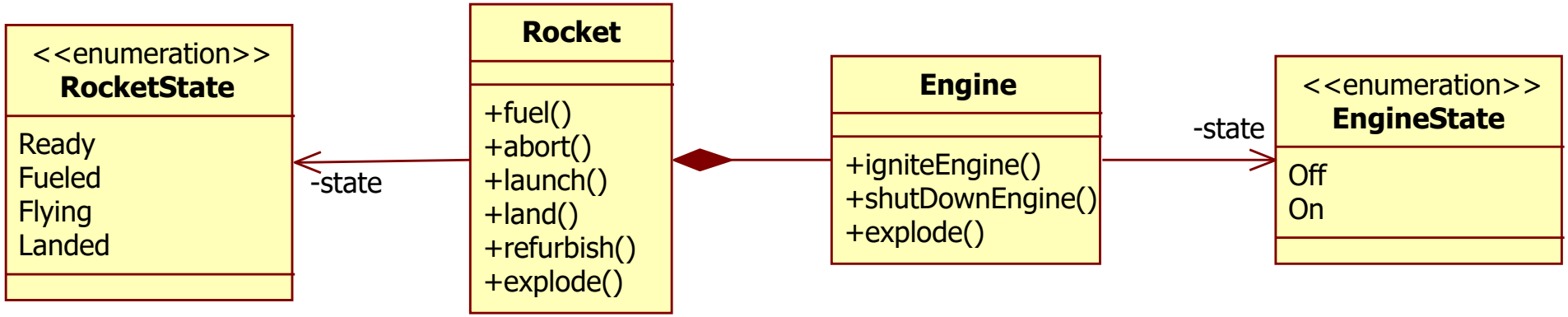


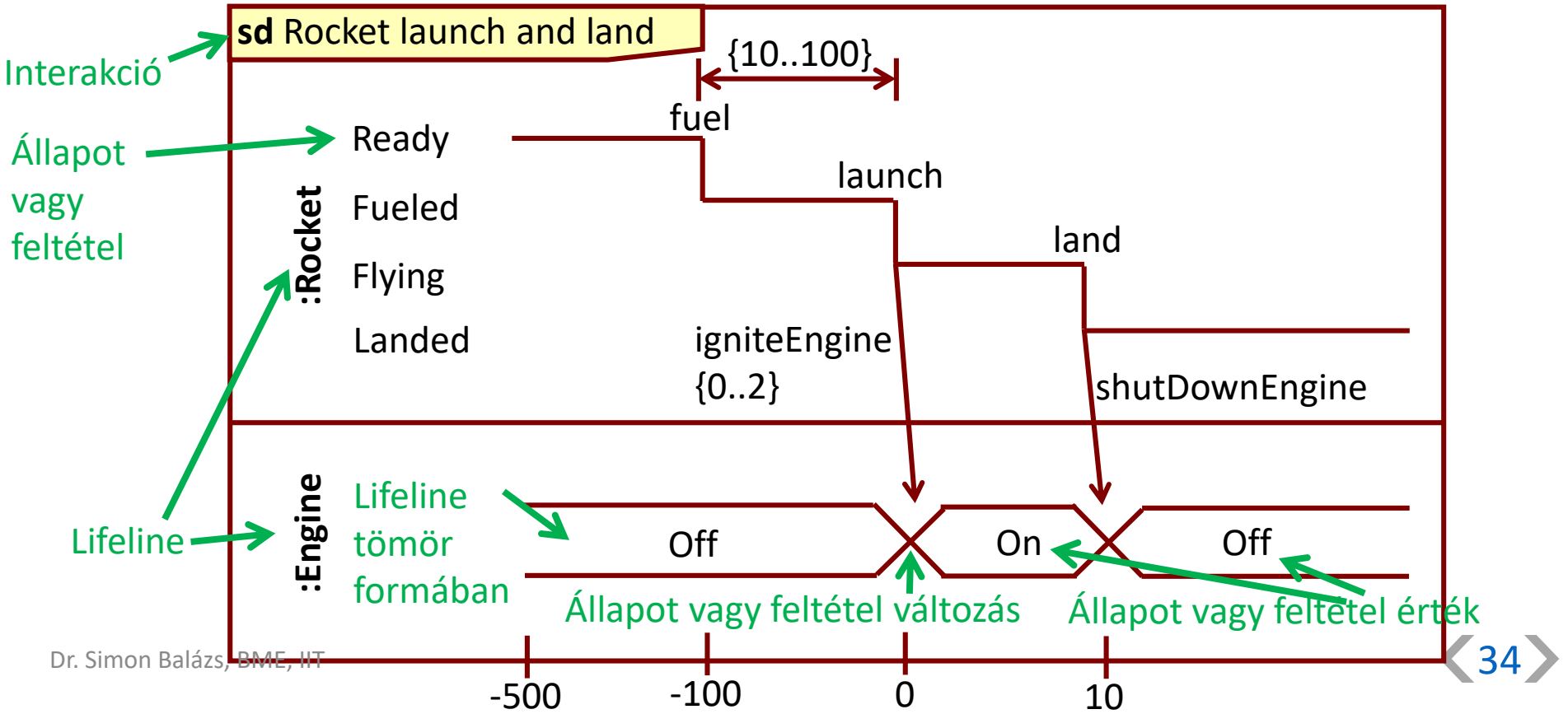
Időzítődiagram példa: rakéta kilövés és leszállás



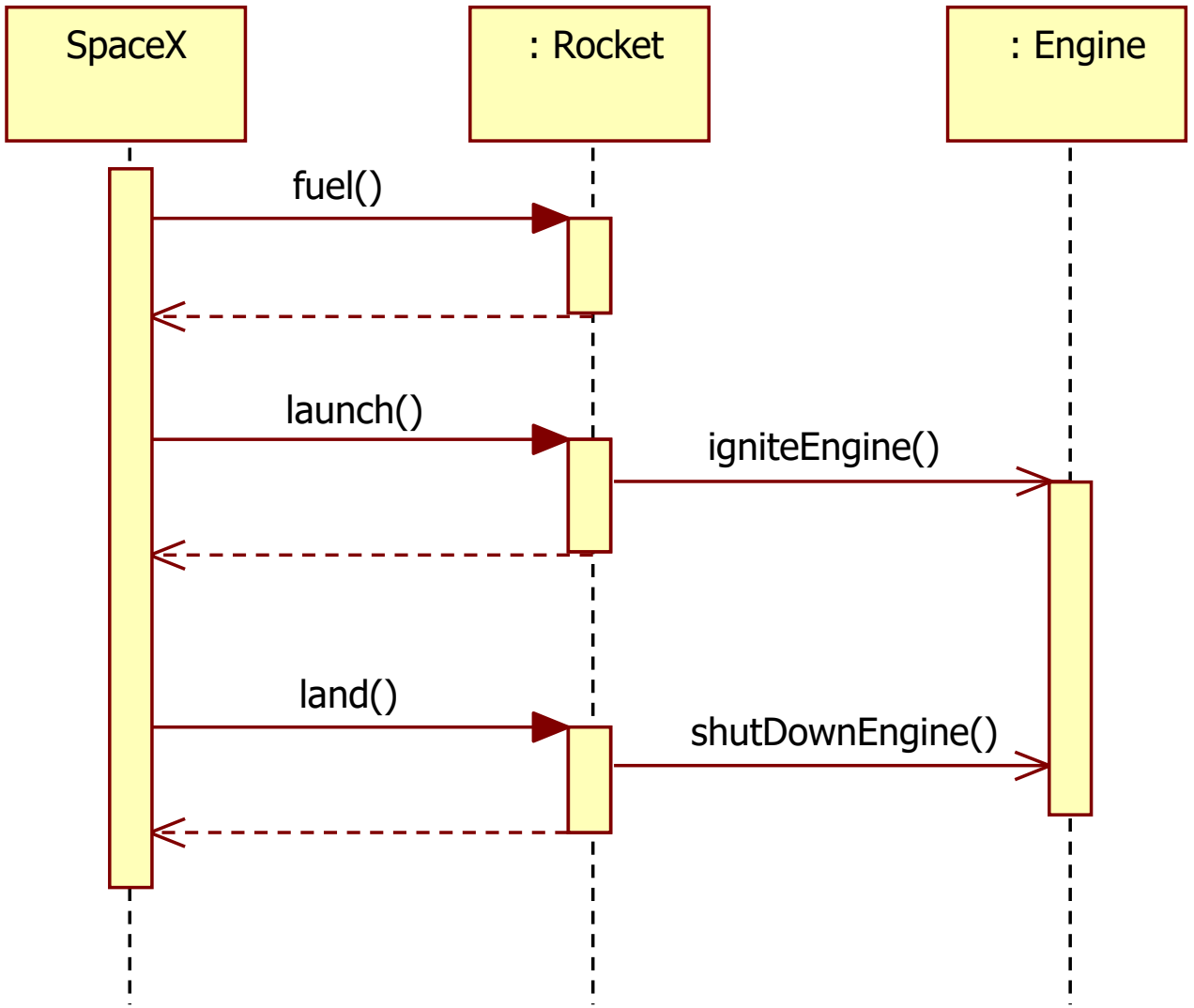


Időzítődiagram példa: rakéta kilövés és leszállás

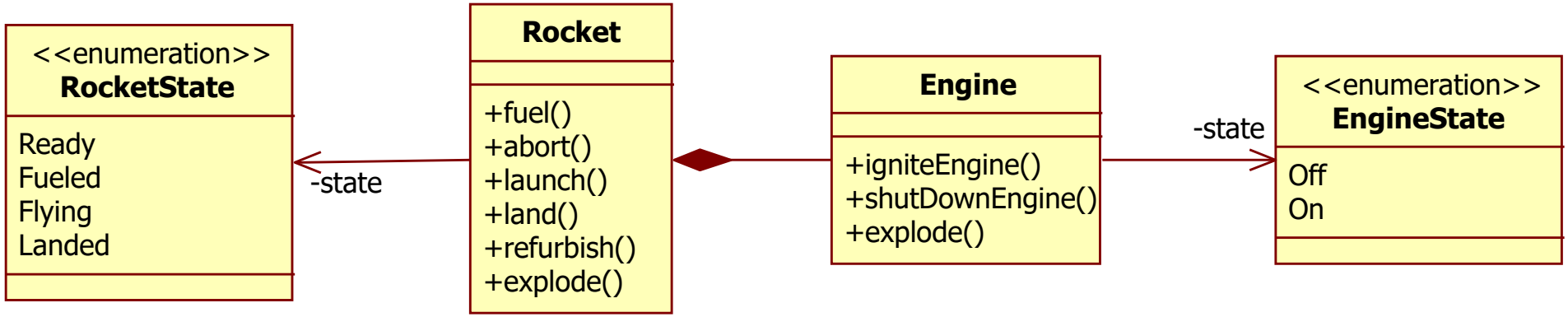


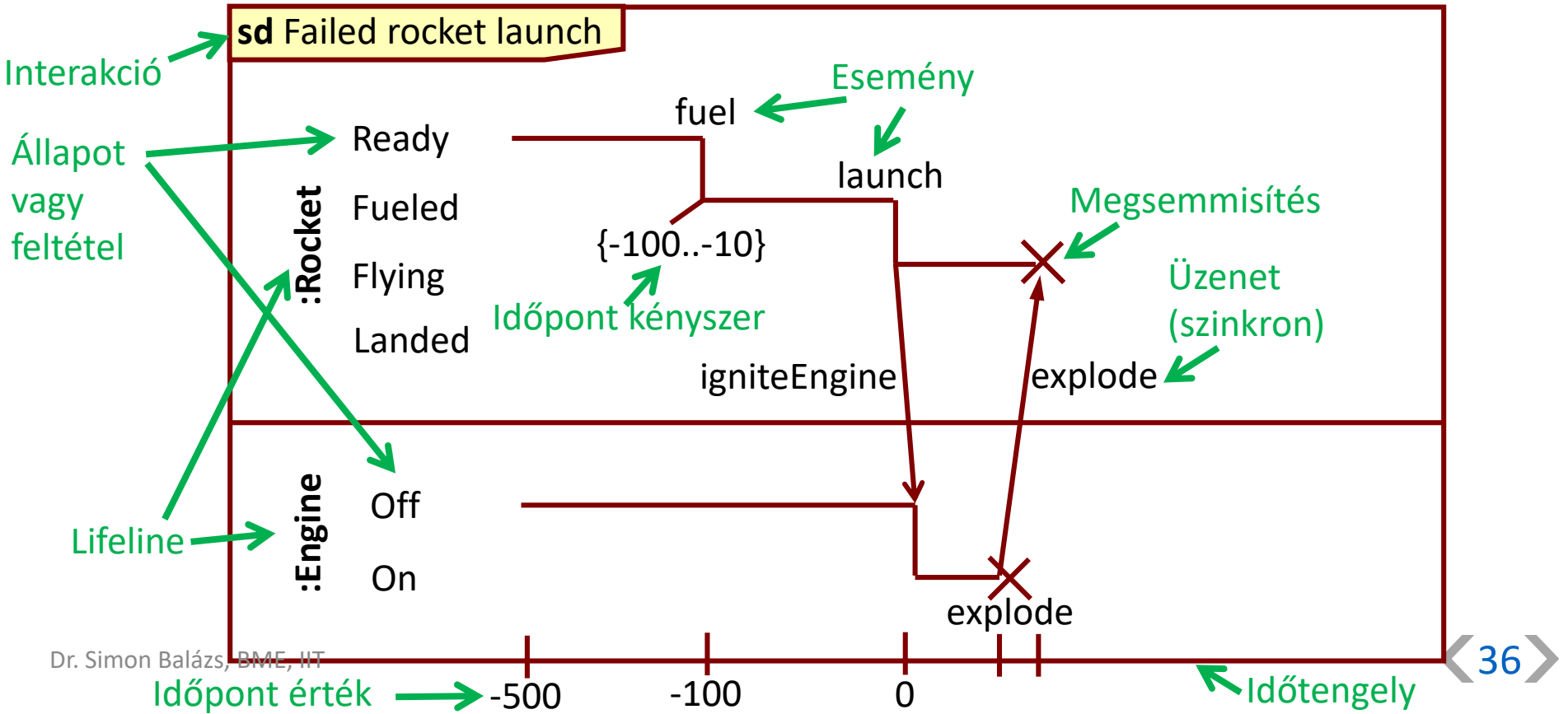


Szekvenciadiagram példa: rakéta kilövés és leszállás

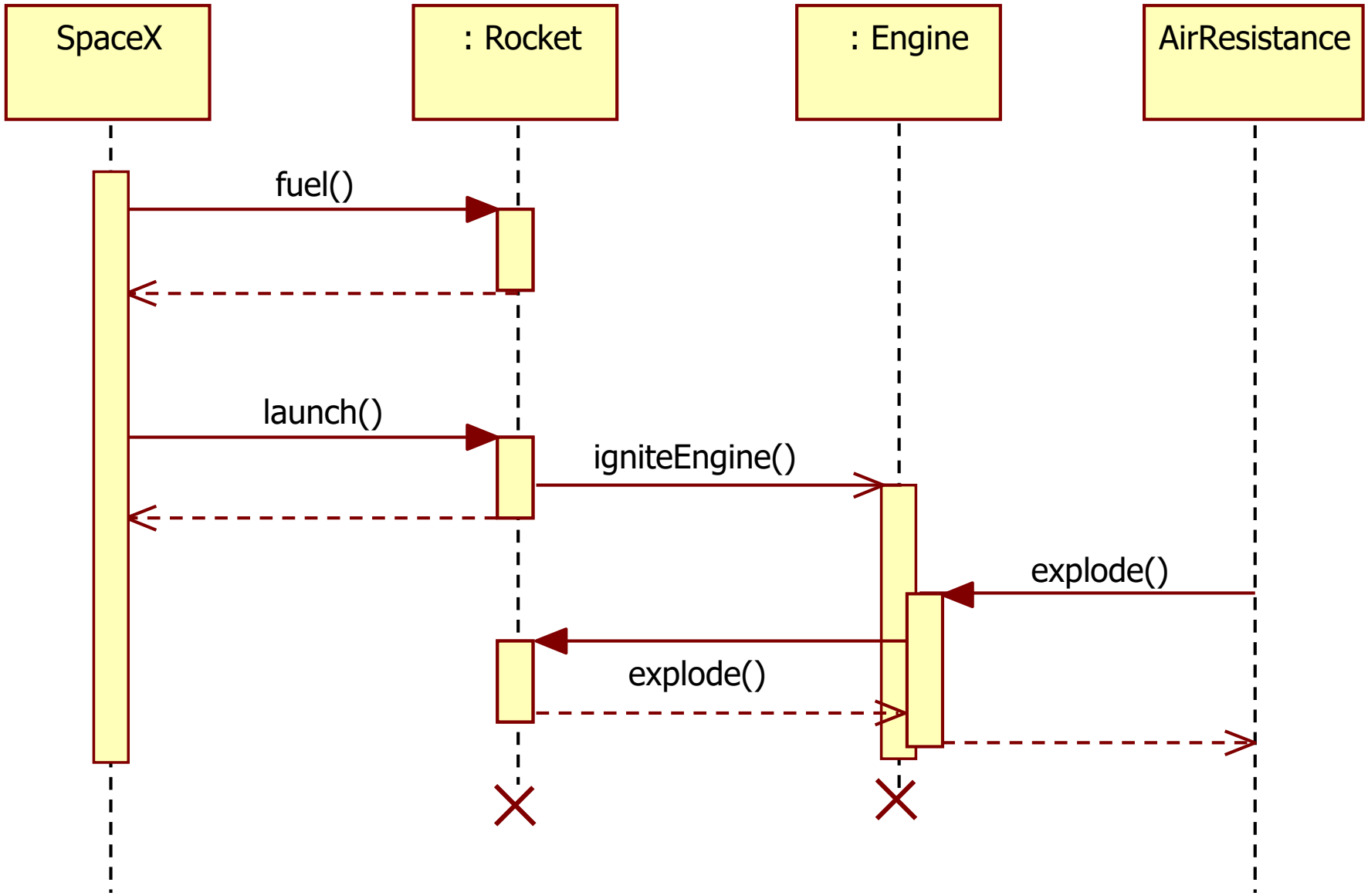


Időzítődiagram példa: sikertelen rakéta kilövés





Szekvenciadiagram példa: sikertelen rakéta kilövés



Hol tartunk?

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

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

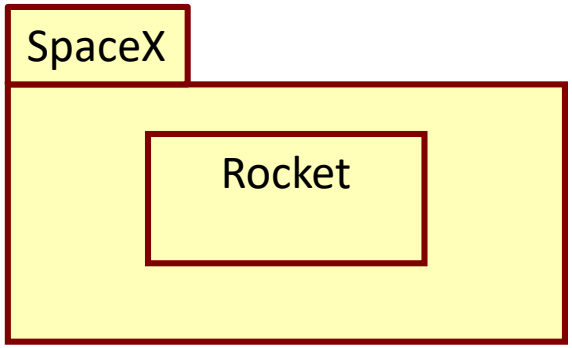
Összetett struktúradiagram (Composite Structure Diagram)

Összetett struktúradiagram (Composite Structure Diagram)

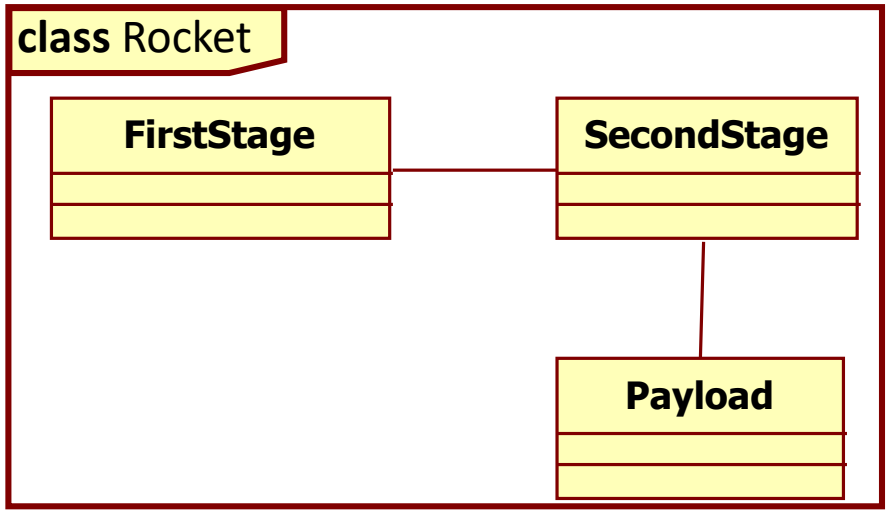
- Egy struktúrával rendelkező classifier (pl. osztály, komponens) belső szerkezetét mutatja
- Általában egy classifier belső szerkezetét nem mutatjuk egy komponensdiagramon vagy osztálydiagramon
  - ilyenkor hasznos az összetett struktúradiagram

Összetett struktúradiagram példa

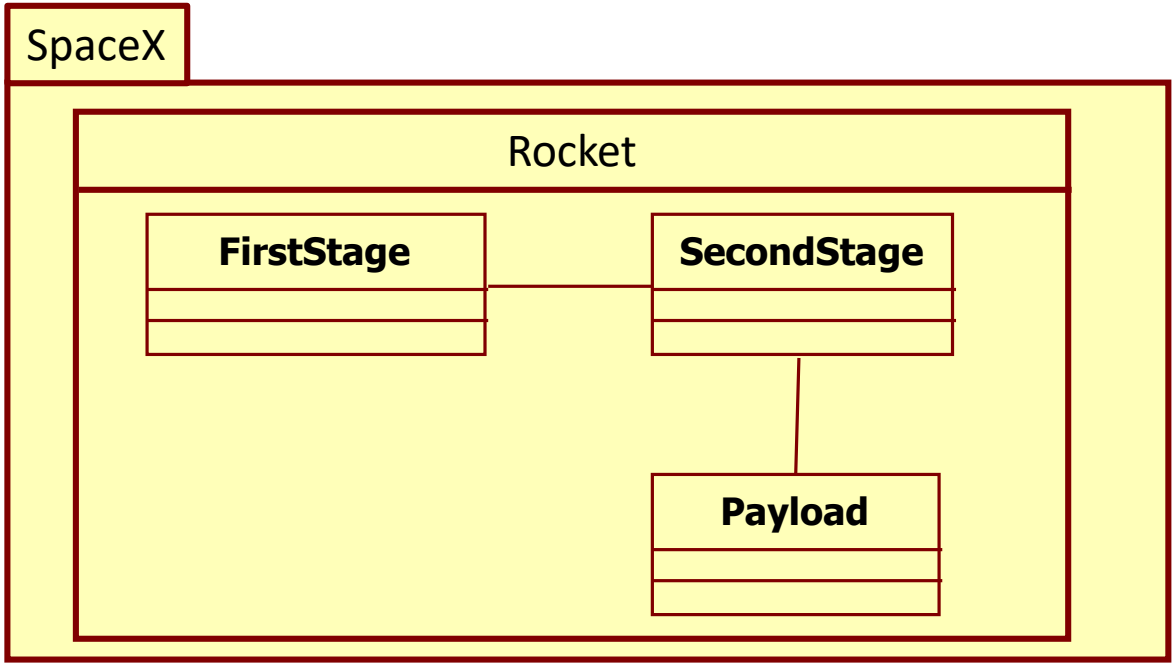
Osztálydiagram, ahol a Rocket osztály belső szerkezete nem látszik:



Összetett struktúradiagram a Rocket osztályra:



Osztálydiagram, ahol a Rocket osztály belső szerkezetét külön compartment mutatja:



Hol tartunk?

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

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

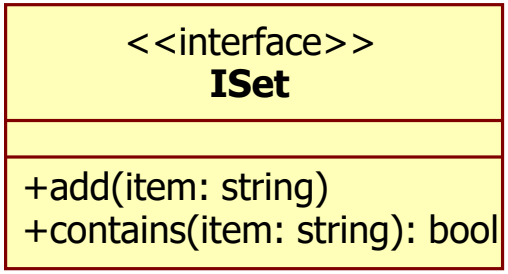
Profildiagram
(Profile Diagram)

Profildiagram (Profile Diagram)

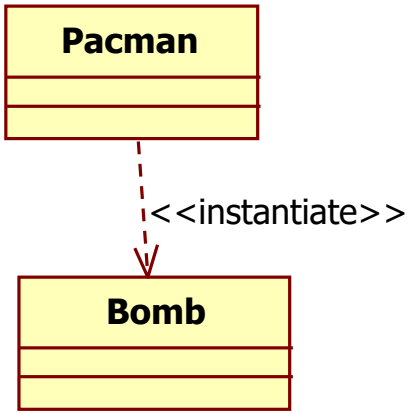
- A profildiagram segítségével saját sztereotípiákat/kulcsszavakat (stereotype/keyword) definiálhatunk
- A sztereotípiák modellelemekhez csatolhatók
- A sztereotípiák módosítják az adott modellelem jelentését
  - az egyénileg definiált sztereotípiák jelentését nem az UML szabvány határozza meg, hanem mi
  - az egyéni sztereotípiák értelmezése és feldolgozása a mi feladatunk, amikor az UML diagramokból programkódot készítünk
- A sztereotípiák kiváló bővítési lehetőséget biztosítanak az UML-ben

Szabványos sztereotípiák

- Vannak sztereotípiák, amelyeket az UML szabvány definiál
- Példák:
  - **<<interface>>**
    - egy osztályhoz csatolható
    - azt jelzi, hogy ez többé már nem osztály, hanem egy interfész



  - **<<instantiate>>**
    - egy függőséghez csatolható
    - pontosítja a függőség jelentését: azt jelzi, hogy a kliens példányokat készít a szerverből

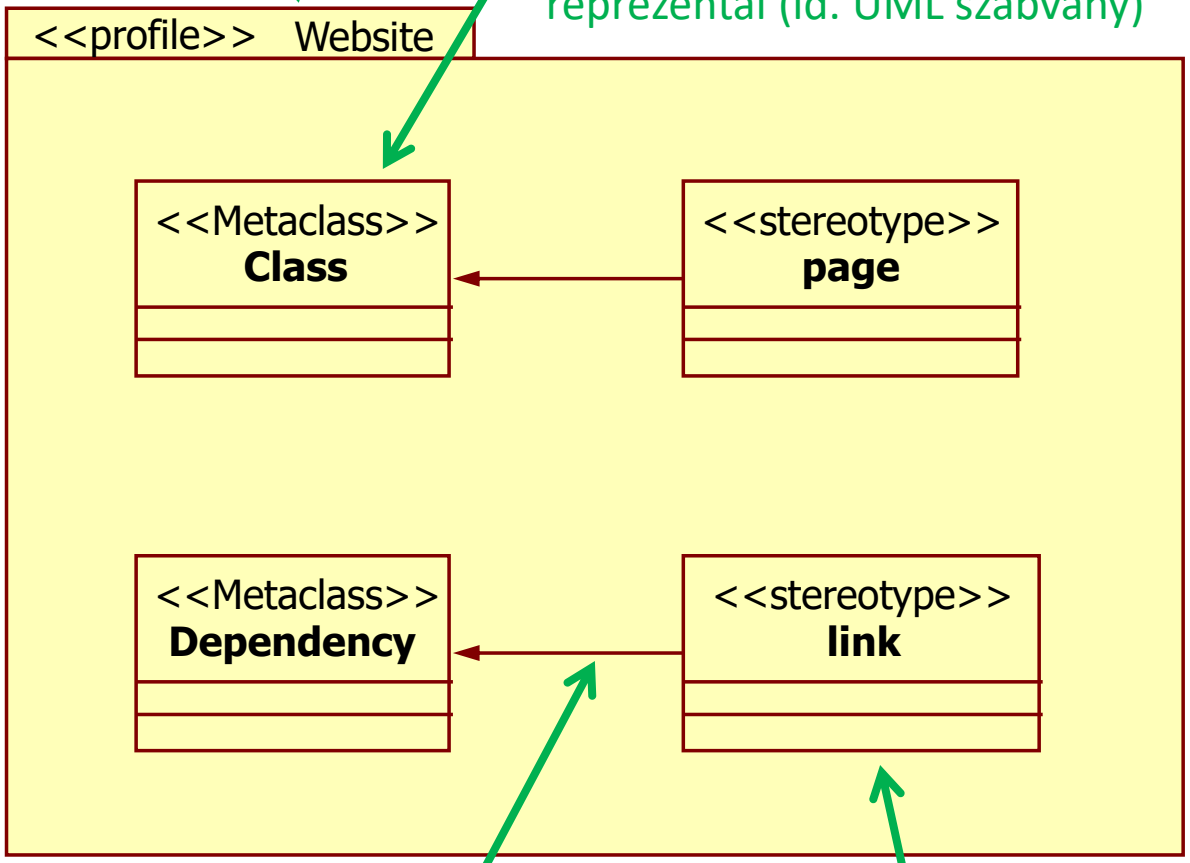


- De készíthetünk saját sztereotípiákat is a profildiagramok segítségével

Profildiagram példa: Weboldalak

Egy <<profile>> sztereotípiával ellátott csomag egy UML profilt definiál

Egy <<MetaClass>> sztereotípiával ellátott osztály egy UML szabványban definiált modellelemet reprezentál (ld. UML szabvány)

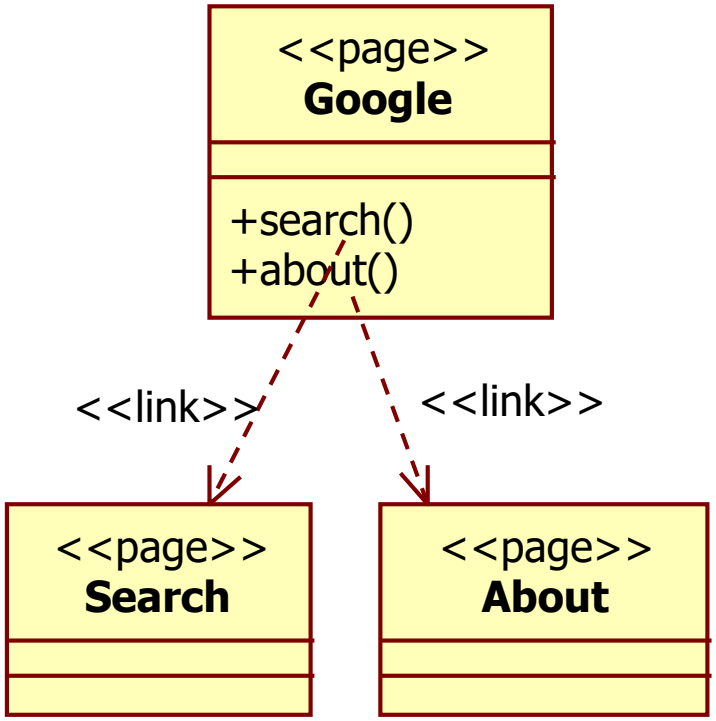


Ez a nyíl azt jelzi, hogy az adott sztereotípia hozzárendelhető az adott modellelemhez

Egy <<stereotype>> sztereotípiával ellátott osztály egy egyéni sztereotípiát definiál

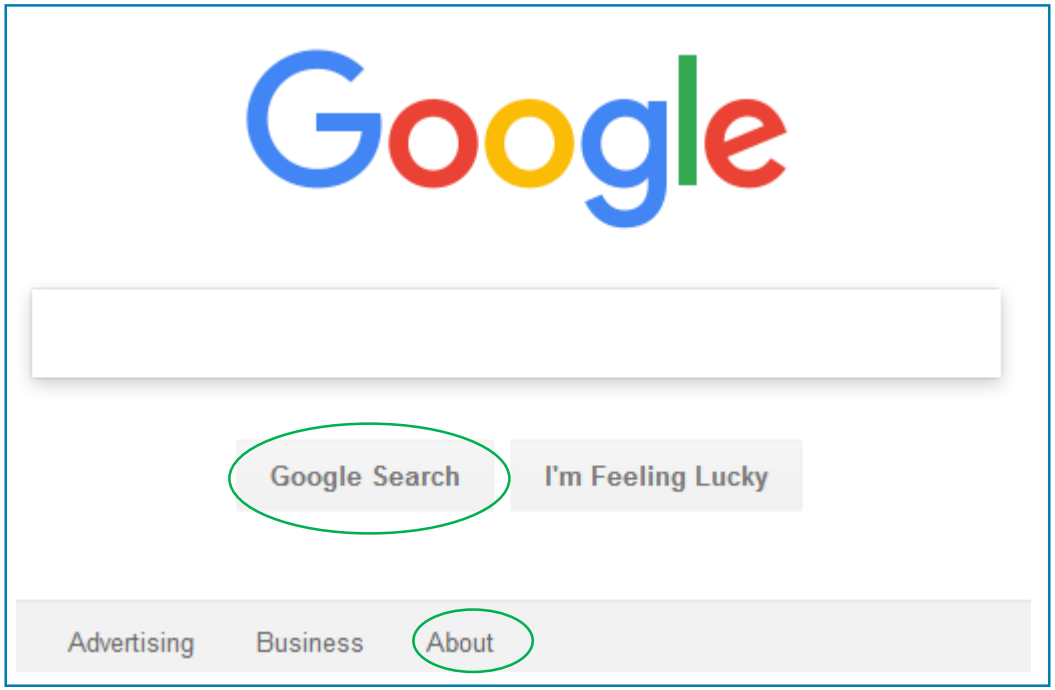
Osztálydiagram példa: Weboldal

Egy osztálydiagram, amely az általunk definiált egyéni sztereotípiákat használja:



A diagram értelmezése rajtunk múlik, mi definiáljuk a sztereotípiák jelentését

Például a diagram jelentheti azt, hogy minden <<page>> sztereotípiával rendelkező osztály egy weboldal, és minden <<link>> -kel rendelkező függőség egy link a következő oldalra:



A sztereotípiák nagyon jó bővítési lehetőséget adnak az UML-hez.
Egész diagramok vagy modellelemek jelentését átdefiniálhatjuk!

Hol tartunk?

Strukturális UML diagramok:

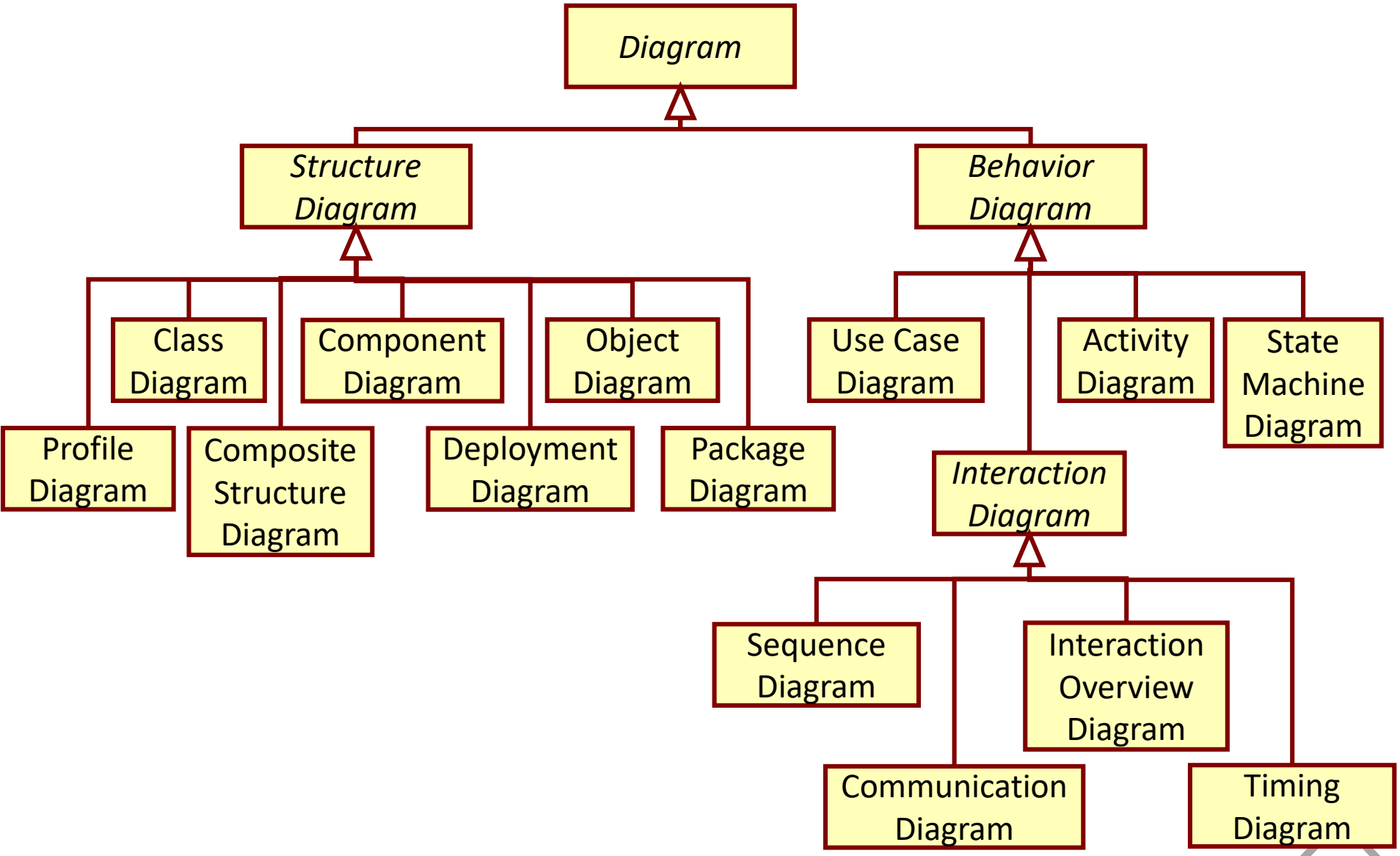
Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

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

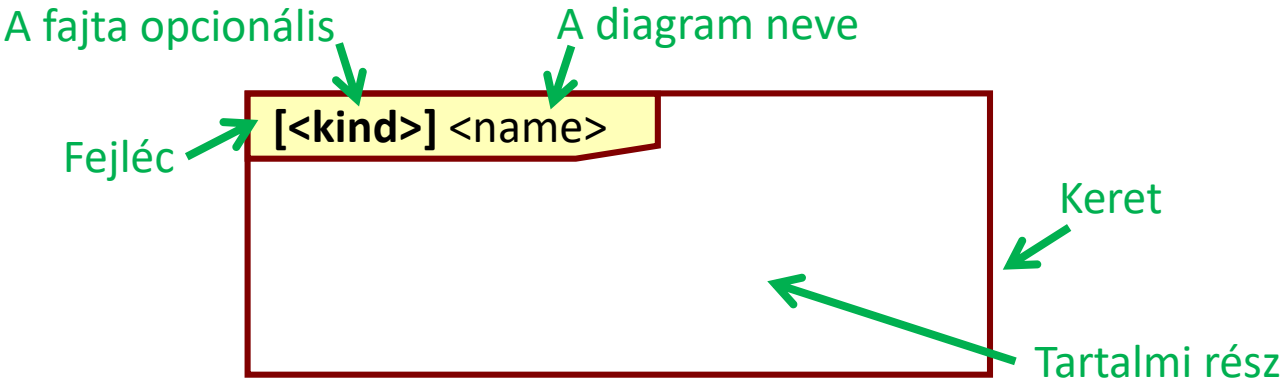
Az UML diagramok összefoglalása

UML diagramok



Keret (frame)

- Minden diagramnak van egy tartalmi része
- Opcionálisan egy diagramnak lehet kerete és fejléce is:



- A keret egy téglalap:
  - elsődlegesen akkor használjuk, ha a diagram által reprezentált elem szélén is lehetnek modellelemek, pl.
    - osztályok és komponensek esetén: portok
    - állapotgépek esetén: belépési és kilépési pontok
    - szekvenciadiagram: kapuk
- Ha nincs rá szükség, a keret elhagyható, és a tervezőeszköz rajzterületének széle alkotja a keretet
  - ha nincs keret, akkor nincs fejléc sem

Diagramok fajtái

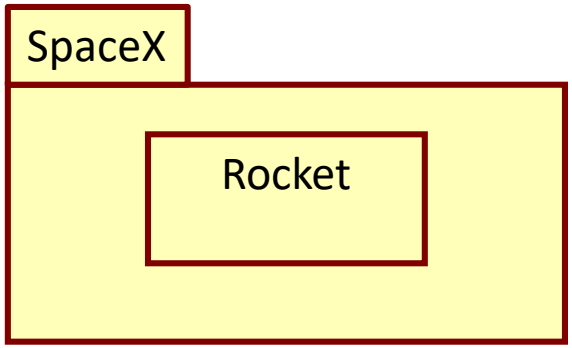
- A fajta az alábbiak egyike:

Fajta	Rövidítés
activity	**act**
class	
component	**cmp**
deployment	**dep**
interaction	**sd**
package	**pkg**
state machine	**stm**
use case	**uc**

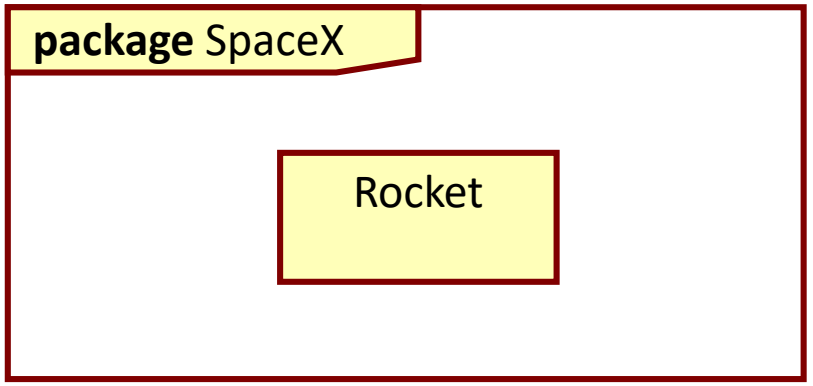
- A fajta hosszú változata helyett a rövidítés is használható

Keret példa

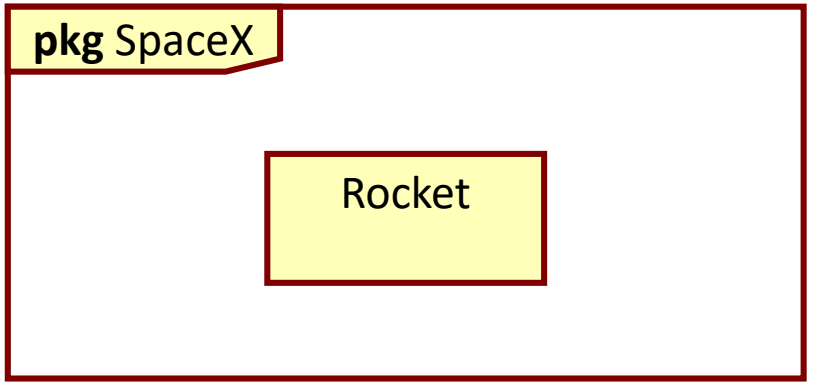
A SpaceX csomag egy nagyobb osztály-diagram részeként ábrázolva:



A SpaceX csomag és tartalma keretként ábrázolva:

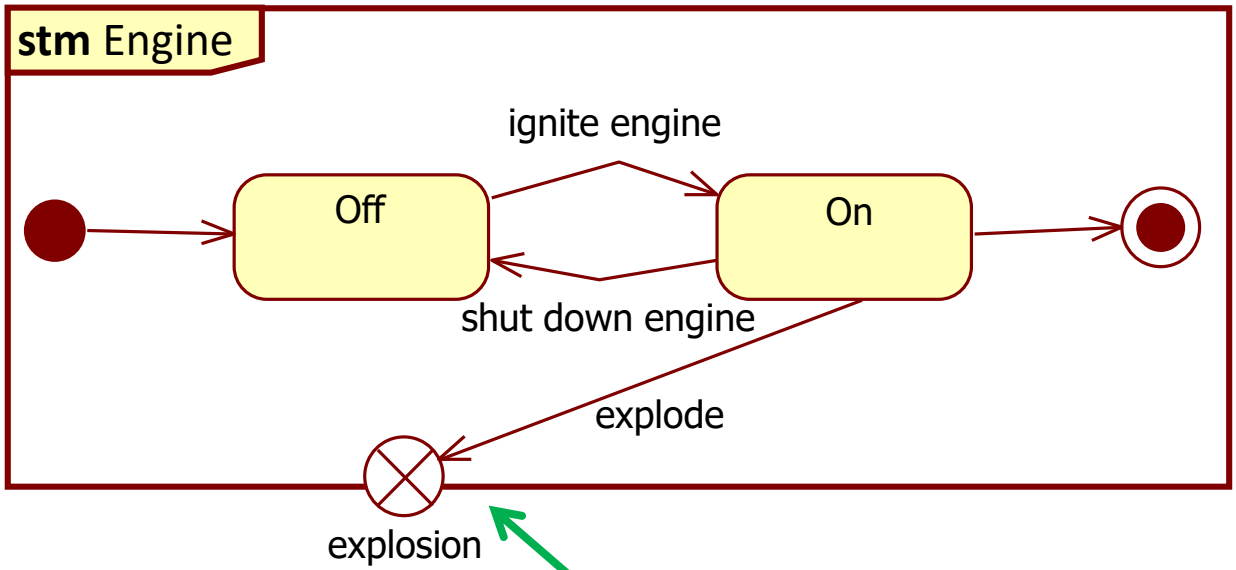


A SpaceX csomag és tartalma keretként ábrázolva:



Keret példa

Állapotdiagram az Engine osztályhoz:



A keret szükséges a kilépési pont miatt

Az UML-en túl

Az UML-en túl

- Az UML-t az Object Management Group (OMG) definiálta
- Az OMG egyéb UML-hez köthető szabványokat is definiál:
  - **Object Constraint Language (OCL)**
    - szöveges szkriptnyelv, amely segítségével metódusok viselkedése, azok előfeltételei és utófeltételei, valamint osztályok invariáns tulajdonságai is leírhatók
  - **XML Metadata Interchange (XMI)**
    - modellezőeszközök között diagramok cseréjére szolgál
  - **MetaObject Facility (MOF)**
    - az UML szabvány definiálására használt modellező nyelv
    - a MOF az UML egy részhalmaza: egy egyszerűsített osztálydiagram
    - a MOF más modellező nyelvek leírására is használható
    - a MOF saját magát is le tudja írni: a MOF a MOF segítségével van definiálva

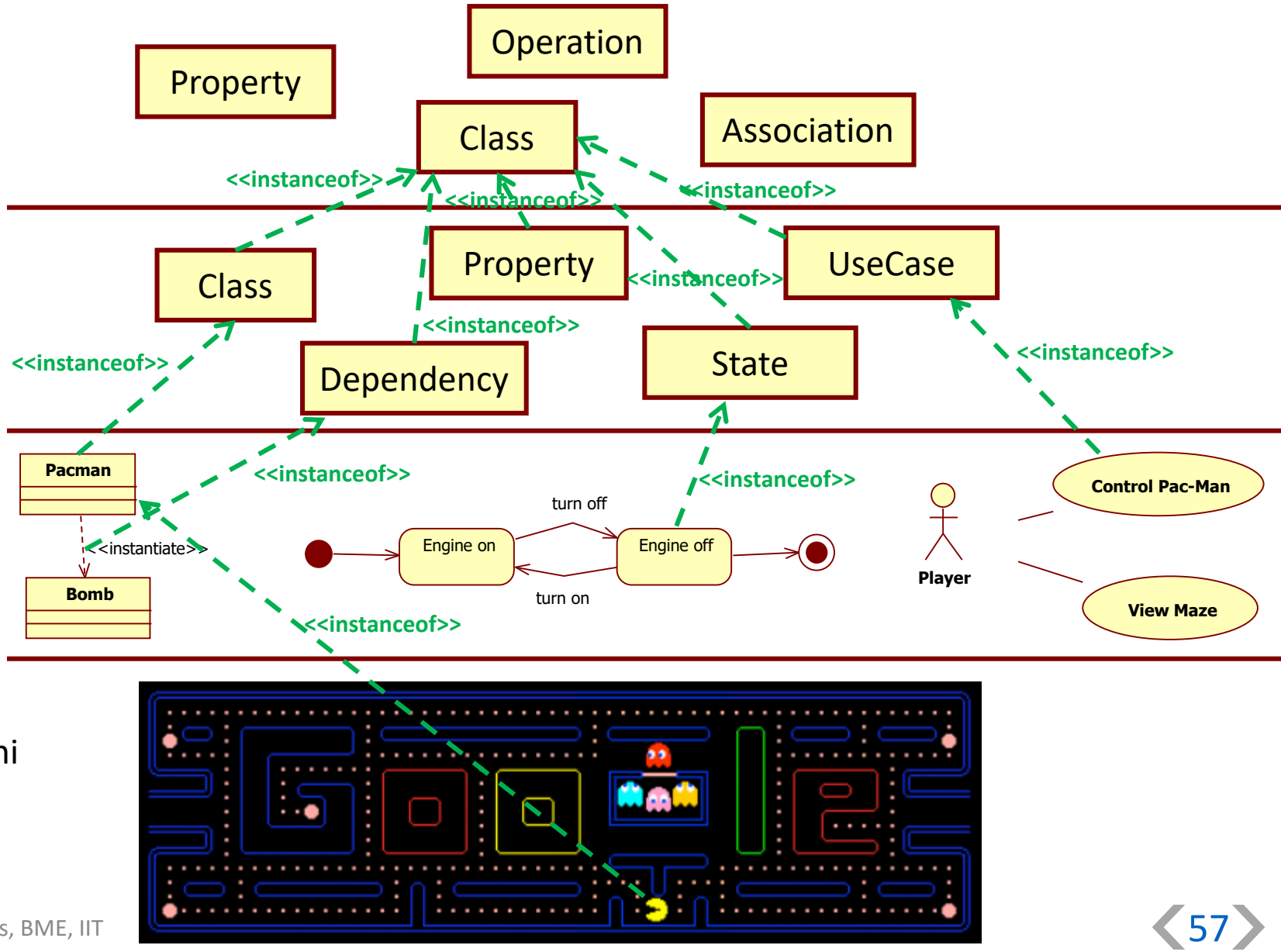
Meta szintek

MOF

UML szabvány

Általunk készített UML modellek

Futás közbeni objektumok

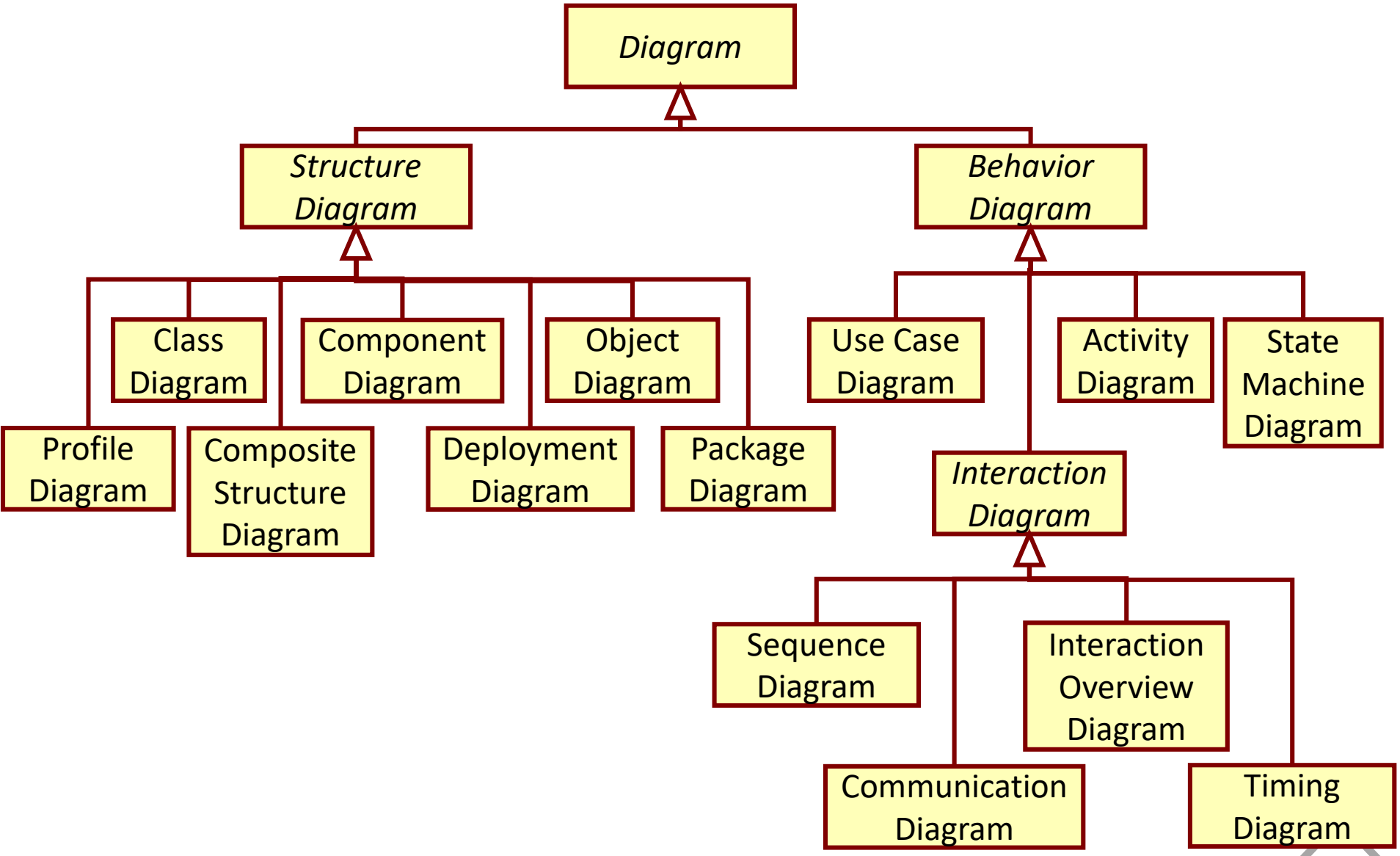


Összefoglalás

Összefoglalás

- UML diagramok:
  - Állapotdiagram
  - Időzítődiagram
  - Összetett struktúra diagram
  - Profildiagram
- Az UML diagramok összefoglalása
- Az UML-en túl:
  - Object Constraint Language (OCL)
  - XML Metadata Interchange (XMI)
  - MetaObject Facility (MOF)

UML diagramok



Hol tartunk?

Strukturális UML diagramok:

Komponens-diagram	Telepítési diagram	Osztálydiagram	Csomagdiagram
Objektumdiagram	Összetett struktúradiagram	Profildiagram	

Viselkedési UML diagramok:

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

Objektumorientált tervezési elvek

Szoftvertechnológia

Dr. Simon Balázs

BME, IIT

Tartalom

- OO fogalmak
- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Don’t Repeat Yourself (DRY)
- Single Choice Principle (SCP)
- Tell, Don’t Ask (TDA)
- Law of Demeter (LoD)

OO fogalmak

OO fogalmak

- Osztály: **típus (type)**
  - definiálja egy objektum metódusait/függvényeit (**viselkedését**, szolgáltatásait)
  - a mezők/attribútumok/változók a viselkedést támogatják, és az értékeik határozzák meg egy objektum **állapotát**
- Objektum: **példány (instance)**
  - egy osztály egy példánya
- **Statikus tagok (static members)**: osztály szintű tagok
  - ezek a metódusok és mezők az osztályhoz tartoznak
  - csak egyetlen példány van belőlük, amely minden objektumra közös
  - statikus függvényeknek nincs this pointere, nem tudják közvetlenül elérni a példány szintű tagokat
- **Példány tagok (instance members)**: objektum szintű tagok
  - ezek a metódusok és mezők az objektumokhoz tartoznak
  - objektumonként külön példány van belőlük
  - példány metódusoknak közös az implementációja, de van egy implicit nulladik paraméterük: a **this** pointer (az aktuális objektum)

OO 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 mezőihez
  - 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

OO fogalmak

- Láthatóság:
  - **private** (-): csak az adott osztályon belül elérhető
  - **protected** (#): csak az adott osztály és leszármazottai számára elérhető
  - **public** (+): bárki számára elérhető, aki az osztályt ismeri
  - **package** (~): az osztály csomagján belül elérhető
- **Virtuális (virtual) metódus**:
  - virtuális függvények felüldefiniálhatók (override) a leszármazott osztályokban
  - így lehet kiterjeszteni (extend) az ős viselkedését
- **Absztrakt (abstract) metódus**:
  - implementáció nélküli virtuális függvény
- **Absztrakt (abstract) osztály**:
  - absztrakt osztály **nem példányosítható**
  - általában van legalább egy absztrakt függvénye, de ez nem szükséges feltétel
- **Interfész (interface)**:
  - függvények halmaza
  - definiálja az interfészt implementáló osztályoktól elvárt viselkedést (szerződést)
- **Osztály interfésze**:
  - az osztály publikus függvényeinek halmaza

OO fogalmak

- **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

OO tervezési elvek

Változás

- Egy szoftver folyamatosan változik
- A jól megtervezett szoftvert könnyű változtatni
- Akkor van gond, ha a követelmények úgy változnak, hogy azokat nehéz beépíteni a szoftverbe
- Ha a szoftver nem tudja követni a változó követelményeket, akkor az a szoftver tervének hibája
- Későbbi fejlesztések megsérthetik az eredeti tervezési filozófiát, és azután csak eszkalálódnak a problémák
  - ezért fontos a dokumentáció

Rossz terv

- A szoftver rosszul van megtervezve, ha:
  - nehéz rajta változtatni, mert a változtatást a rendszer sok különböző részén el kell végezni (a terv merev)
  - a változások a szoftver olyan részeit is elrontják, amelyekre nem számítottunk (a terv törékeny)
  - másik szoftverben nehéz újrahasznosítani az adott szoftver egyes részeit, mert nem lehet leválasztani őket a többi komponenstől (a terv nem mobilis)
- A rossz tervezés oka: túl sok függőség a szoftver egyes részei között
- Megoldás:
  - csökkentsük a részek közötti függőségeket
  - változtassuk meg a függőségek irányát úgy, hogy ne a problémás és gyakran változó részek felé mutassanak

SOLID elvek

- A jó OO tervezés öt alapelve:
  - Single responsibility (egyetlen felelősség elve)
  - Open-closed (nyílt-zárt elv)
  - Liskov substitution (Liskov-féle helyettesíthetőség elve)
  - Interface segregation (interfészek szétválasztásának elve)
  - Dependency inversion (függőségek megfordításának elve)
- Robert C. Martin vezette be ezeket
- Ezek az elvek elősegítik a későbbi karbantarthatóságot és bővíthetőséget azáltal, hogy csökkentik a szoftver egyes részei közötti függőségeket

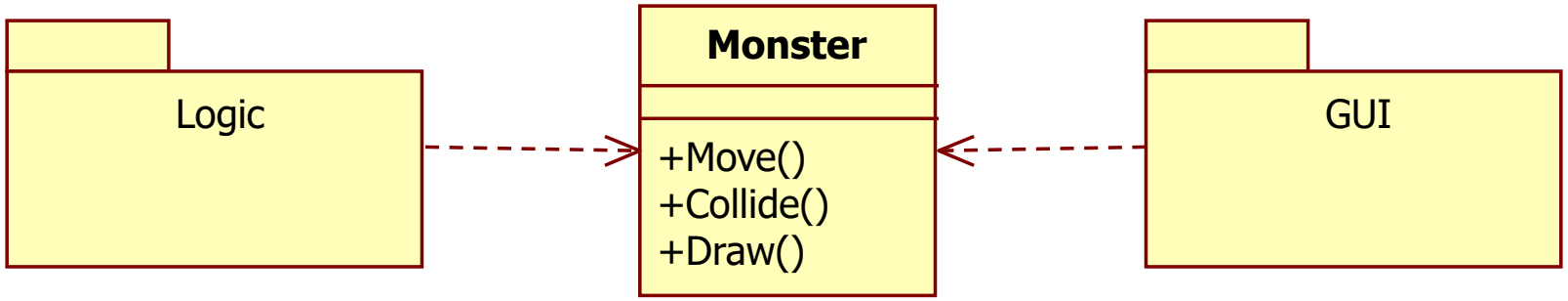
Single Responsibility Principle (SRP)

- Egy osztálynak csak egy oka legyen a változásra
  - (Robert C. Martin)
- Felelősség = ok a változásra (≠ a biztosított szolgáltatások)
- Vagyis: ha egy osztály egynél több felelősséggel rendelkezik, akkor az osztályt több osztályra kéne szétdarabolni

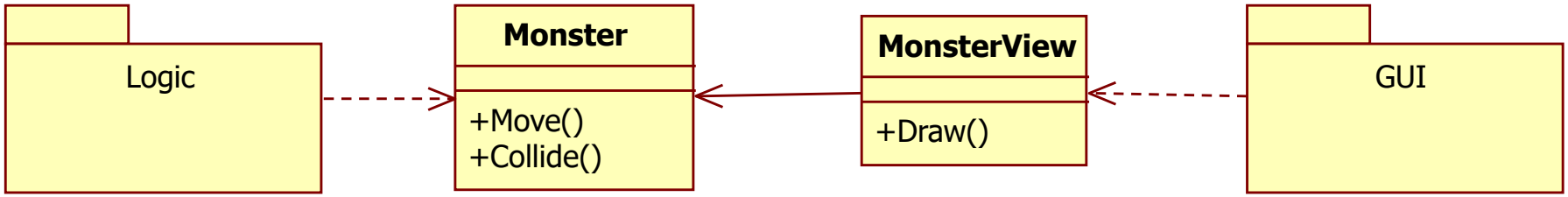
SRP

- Ha egy osztálynak több felelőssége van, válasszuk szét a felelősségeket:
  - implementációs szinten (ha szét lehet őket választani)
  - interfész szinten (ha nem lehet őket szétválasztani)
- Implementációs szinten:
  - külön osztályok
  - körkörös függőségek nélkül
- Interfész szinten:
  - felelősségekként külön interfészek
  - az eredeti osztály implementálja az interfészeket
- Előny: a függőségek a problémás felelősségektől elfelé mutatnak

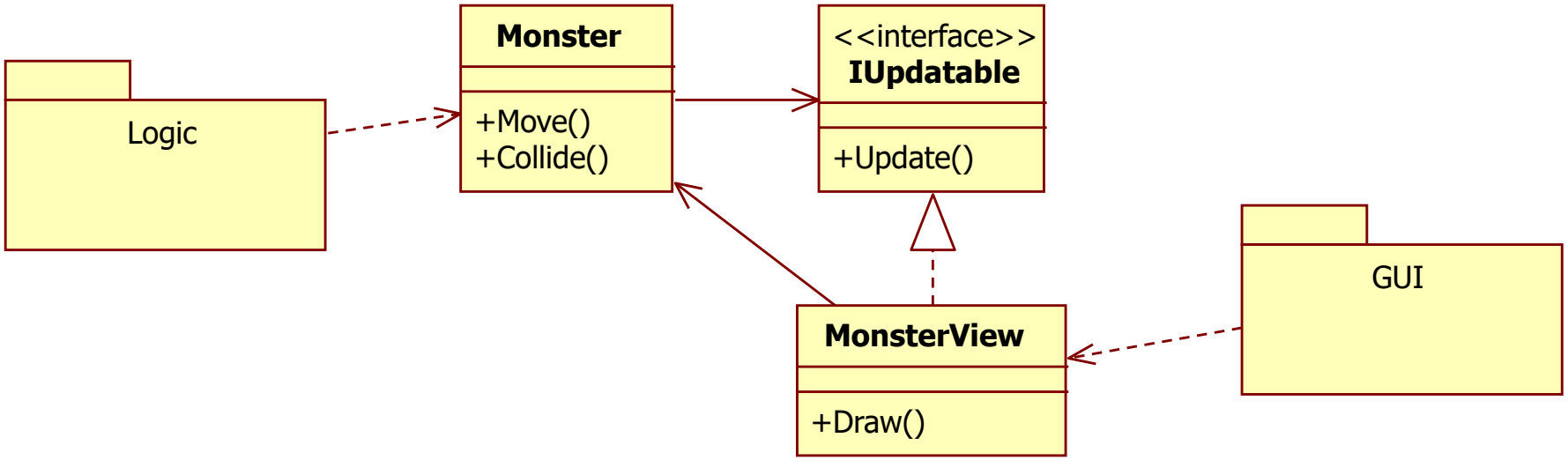
Példa az SRP megsértésére



SRP megoldás I.



SRP megoldás II.



Változás valószínűsége

- Nem mindig egyértelmű, hogy több ok is lehet a változásra
- Lehet, hogy a változási igény sosem következik be
- A tervezéskor meg kell becsülni a változás bekövetkezésének valószínűségét
  - múltbeli tapasztalataink és a szakterülettel kapcsolatos ismereteink alapján
- Feleslegesen ne tervezzünk olyan változásra, amelynek bekövetkezése nagyon alacsony valószínűségű
  - YAGNI = You Ain't Gonna Need It

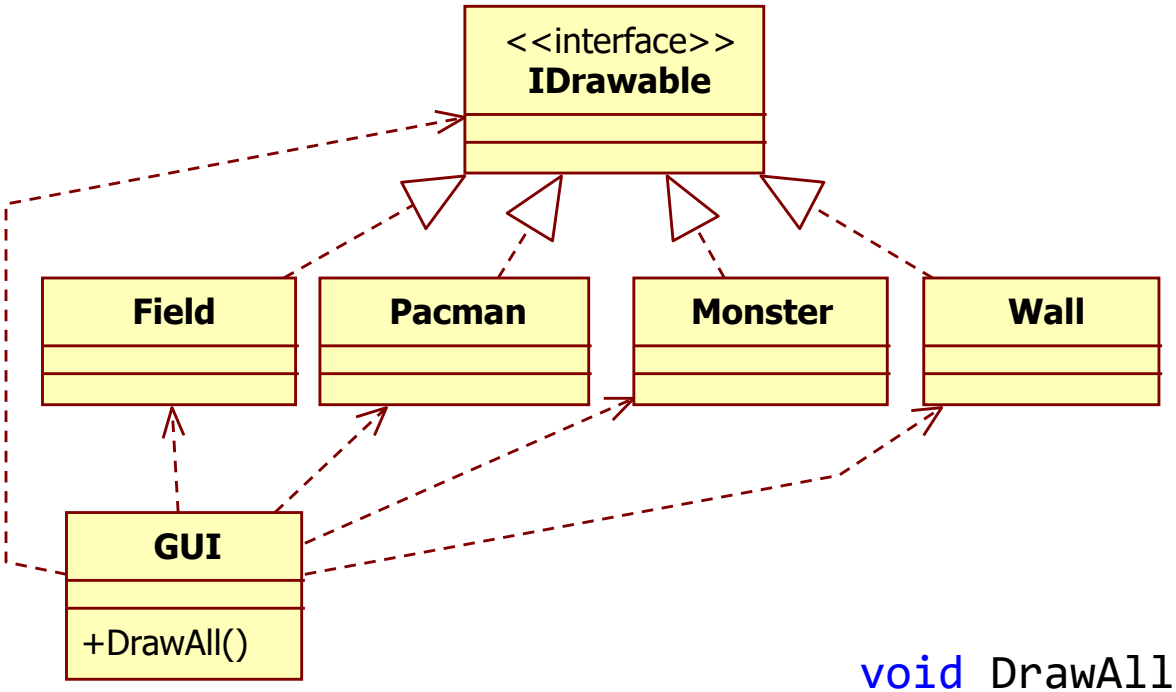
Open/Closed Principle (OCP)

- A szoftver részeinek (osztályok, modulok, függvények, stb.) nyitottnak kell lennie a kiterjesztésre, de zártnak a módosításra
  - (Bertrand Meyer)
- Nyitott a bővítésre: a modul viselkedése kiterjeszthető, hogy megfeleljünk a változó követelményeknek
- Zárt a módosításra: a modul kiterjesztése nem igényelheti a már meglévő kódok átírását

OCP

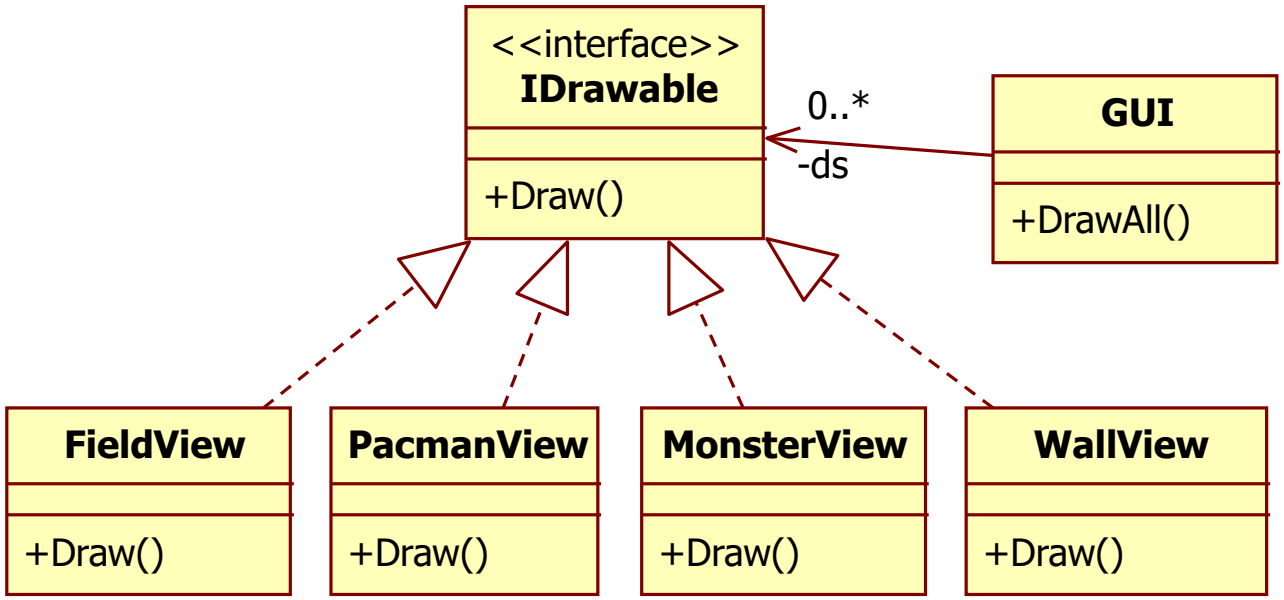
- Készüljünk fel a változásokra
- Nyitott a bővítésre:
  - új leszármazott osztályok
  - metódusok felüldefiniálása
  - polimorfizmus
  - delegáció
- Zárt a módosításra:
  - a már meglévő kód nem változik
  - csak hibajavítások vannak
- A viselkedés kiterjesztése új kód írásával történik, nem a meglévő kód átírásával

Példa az OCP megsértésére




```
void DrawAll(List<IDrawable> ds) {
  foreach (var d in ds) {
    if (d is Field) {
      // ...draw field...
    } else if (d is Pacman) {
      // ...draw pacman...
    } else if ...
  }
}
```

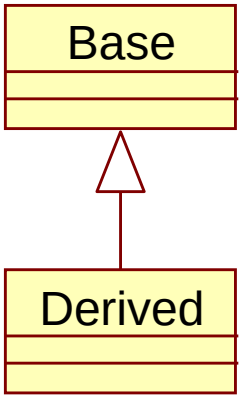
OCP megoldás




```
void DrawAll(List<IDrawable> ds) {
  foreach (var d in ds) {
    d.Draw();
  }
}
```

Liskov Substitution Principle (LSP)

- A leszármazottaknak behelyettesíthetőnek kell lennie az ős típusba
  - (Barbara Liskov)
- Bármely leszármazott (Derived) osztály egy példánya behelyettesíthető egy olyan helyre, ahol az ős (Base) egy példányát használjuk, anélkül, hogy az ős használója észrevenné a különbséget
- Öröklődés:
  - Derived egyfajta Base
  - minden Derived típusú objektum egyben Base típusú is
  - ami igaz a Base-re, igaz a Derived-ra is
  - a Base általánosabb dolgot reprezentál, mint a Derived
  - a Derived egy speciálisabb dolgot reprezentál, mint a Base
  - bárhol ahol a Base használható, egy Derived is használható



LSP

- A Liskov-elv fontossága akkor válik szembetűnővé, ha a megsértésének következményeivel találkozunk
- A Liskov-elv megsértése:
  - a leszármazott nem úgy viselkedik, ahogy azt az őstől elvárnánk
  - tipikusan:
    - öröklés az adatok újrahasznosítása céljából (pl. négyzet-téglalap probléma)
    - paraméterek/visszatérési érték értékkészletének megsértése
- Ennek következményei:
  - a speciális leszármazott felismeréséhez explicit típuslekérdezés szükséges
    - is, instanceof, dynamic_cast, stb.
  - vagyis: az OCP elvet is megsértjük

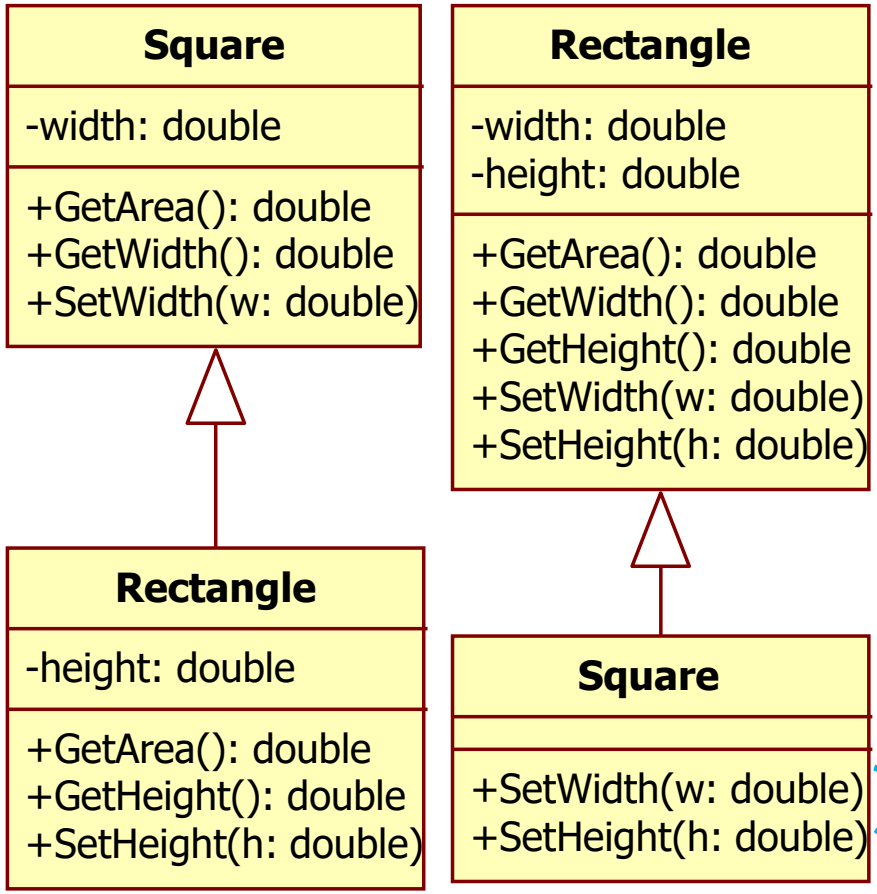
LSP megsértésének következménye: típusellenőrzés

- Ha egy leszármazott megsérti a Liskov-elvet, egy tapasztalatlan fejlesztő sietségében lehet, hogy explicit típusellenőrzéssel oldja meg a problémát
- Például:

```
void Draw(Shape s) {
  if (s is Rectangle) DrawRectangle((Rectangle)s);
  else if (s is Ellipse) DrawEllipse((Ellipse)s);
}
```

  - Itt a Draw megsérti az OCP elvet is, mert a Shape minden leszármazottját ismernie kell
  - A Liskov-elv megsértése általában az OCP megsértését is magával vonja

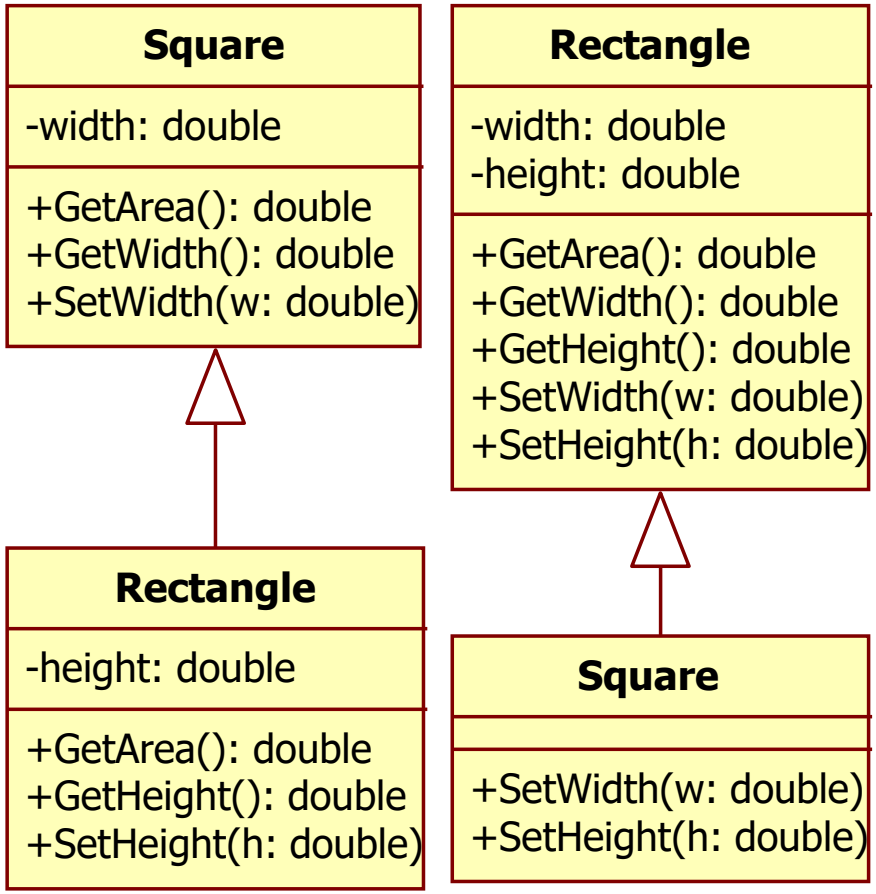
Melyik tervezés jobb?



Egyik sem: mindkettő sérti a Liskov-elvet

```
class Square : Rectangle {
  void SetWidth(double w) {
    base.SetWidth(w);
    base.SetHeight(w);
  }
  void SetHeight(double h) {
    base.SetWidth(h);
    base.SetHeight(h);
  }
  // ...
}
```

Mindkettő sérti a Liskov-elvet




```
void TestSquare(Square s) {
  s.SetWidth(5);
  Debug.Assert(s.GetArea() == 25);
}

void TestRectangle(Rectangle r) {
  r.SetWidth(5);
  r.SetHeight(4);
  Debug.Assert(r.GetArea() == 20);
}
```

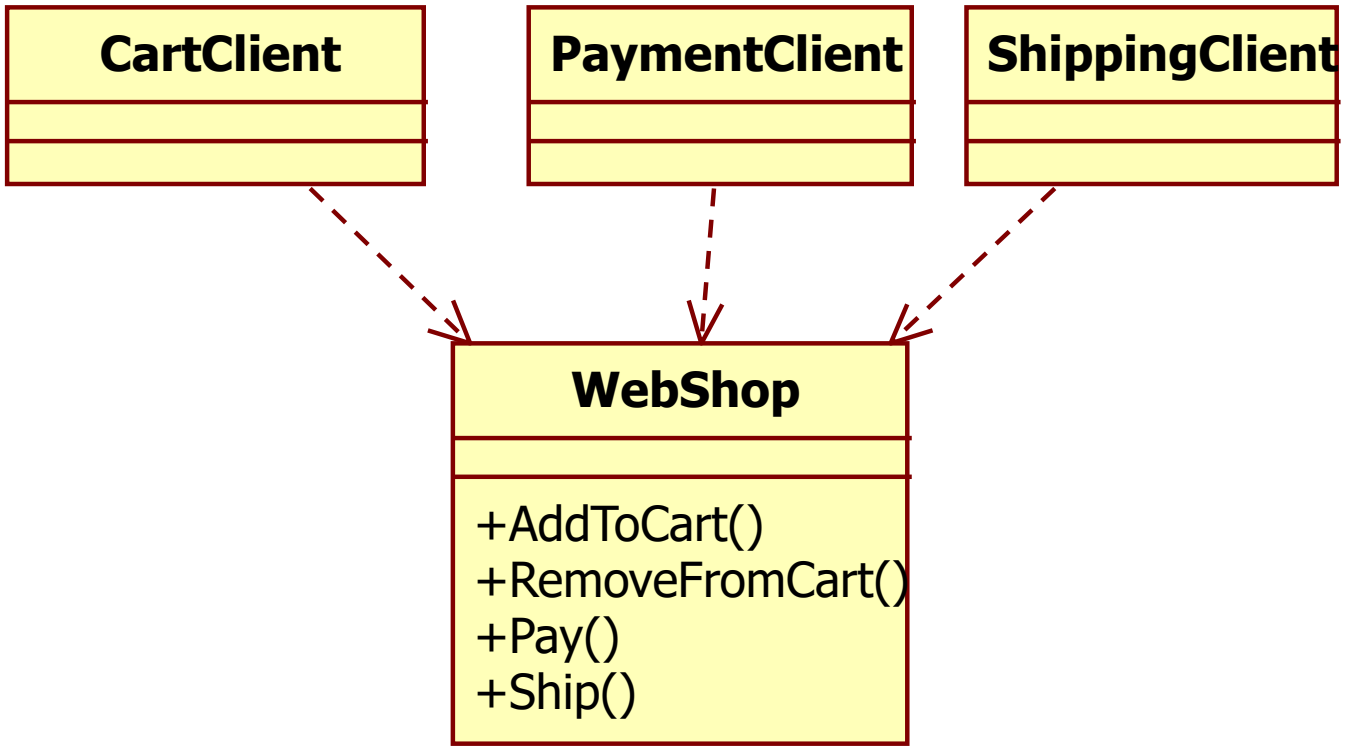
LSP

- A négyzet-téglalap probléma gyökere:
  - a négyzet matematikai (adat) szempontból egyfajta téglalap
  - de a négyzet viselkedése más, mint a téglalapé (a SetWidth-nek nem kéne a magasságot is állítania)
  - és az Objektumorientáltságban a viselkedés számít
- **Fontos**:
  - Az öröklődés nem az adatok újrahasznosítására való!
  - Az öröklődés célja a viselkedés újrahasznosítása!

Interface Segregation Principle (ISP)

- A klienseket nem kötelezhetjük arra, hogy olyan metódusoktól függjenek, amelyeket nem használnak
  - (Robert C. Martin)
- Az ISP elfogadja, hogy egyes osztályoknak nagy és nem kohézív interfésze van szüksége
- De a klienseknek nem szükséges a teljes osztályt ismerni
- Helyette: elég, ha a kliensek csak kohézív interfésszel rendelkező absztrakciókat ismernek

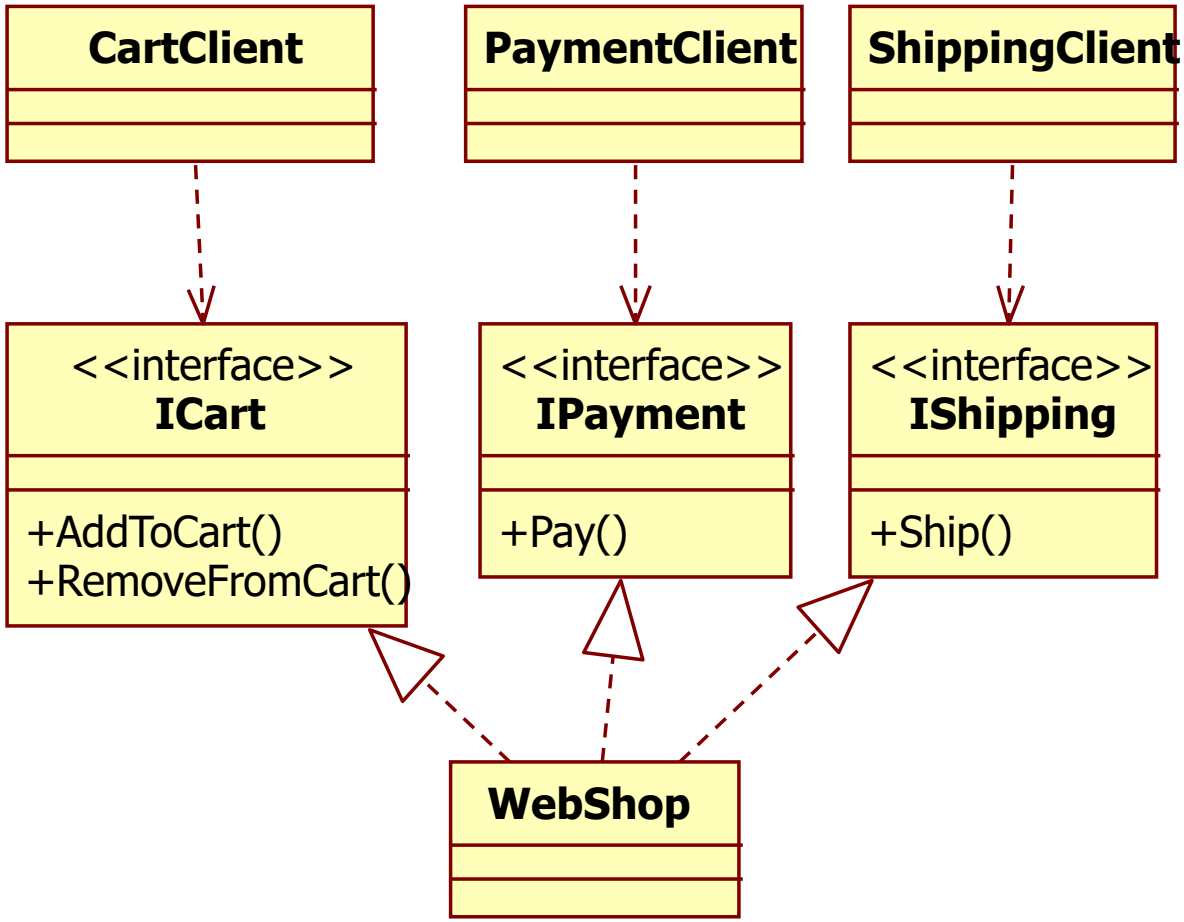
Példa az ISP megsértésére



ISP

- A kliens csak azoktól a metódusoktól függjön, amelyeket ténylegesen meghív
- Daraboljuk fel a nagy kövér osztály interfészét több kliens-specifikus interfészre
- A nagy kövér osztály implementálja ezeket az interfészeket
- A kliensek csak azoktól az interfészektől függjenek, amelyekre szükségük van
- Így a kliensek nem függnek többé a nem használt metódusoktól
- És így a kliensek függetlenek lehetnek egymástól

ISP megoldás

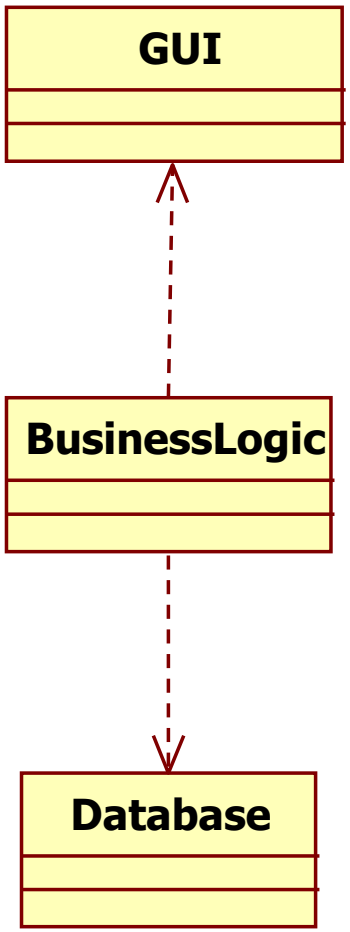


Dependency Inversion Principle (DIP)

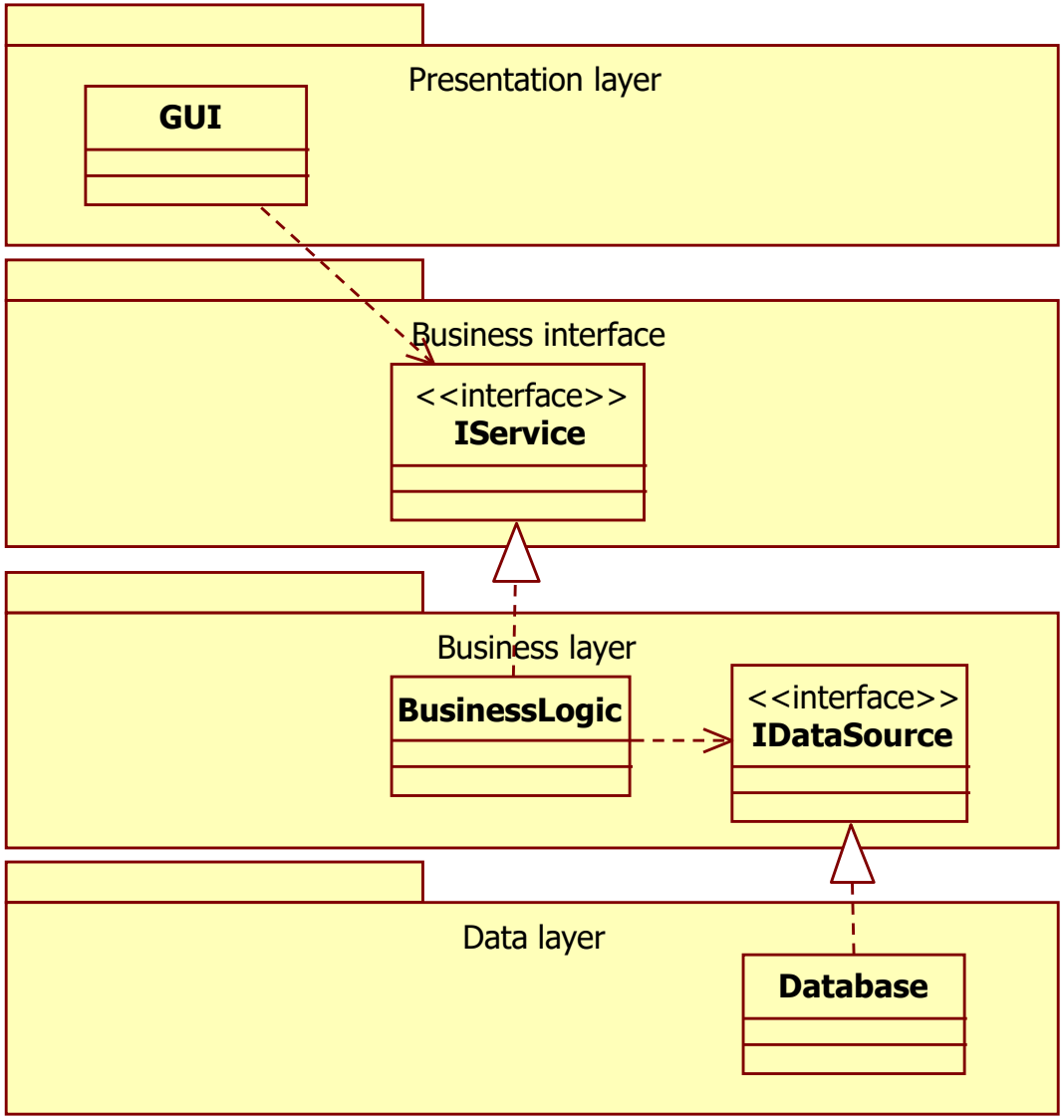
- Magas szintű modulok ne függjenek alacsony szintű moduloktól. Mindkettő absztrakcióktól függjön.
- Absztrakciók ne függjenek a részletektől. A részletek függjenek az absztrakcióktól.
  - (Robert C. Martin)
- Egy alkalmazás modulokból/komponensekből áll
- Egy természetes módja a fejlesztésnek az, hogy megírjuk az alacsony szintű modulokat (input-output, hálózat, adatbázis, stb.) majd beépítjük őket a magasabb szintű modulokba
- Azonban ez rossz: egy alacsony szintű modul változása előidézheti egy magasabb szintű modul megváltozását

Példa a DIP megsértésére

- Példa a rossz tervezésre:
  - Üzleti logika ---> Adatbázis
  - Üzleti logika ---> GUI
  - a konkrét adatbázis technológia vagy konkrét GUI technológia változhat
  - a változás az üzleti logikára is kihathat
- Dependency Inversion Principle:
  - fordítsuk meg a függőség irányát: az alacsony szintű modulok a magasabb szintű modulok által definiált absztrakcióktól függjenek
- A DIP egy másik értelmezése:
  - absztrakcióktól függjünk



DIP megoldás



DIP kritika

- Vannak olyan helyzetek, amikor egy konkrét osztály interfésze változik
- És ennek a változásnak meg kell jelennie az osztályt reprezentáló interfészben
- Egy ilyen változás felszivárog, és megtöri az absztrakt interfész által biztosított izolációt
- DE:
  - a kliens osztály definiálja az interfészt és annak szolgáltatásait, mert ő tudja, mire van szüksége
  - az interfész csak akkor változhat, ha a kliens kezdeményezi a változást!

Don’t Repeat Yourself (DRY)

- Minden tudásnak egyetlen és egyértelmű helyen kell megjelennie a rendszerben
- Csökkentsük az ismétlődést
- A duplikáció rossz:
  - ha az ismétlődő részben valamit meg kell változtatni, akkor azt az összes előfordulási helyen meg kell változtatni
  - nagy a valószínűsége annak, hogy néhány helyen kimarad a változás
- Ha Ctrl+C-t nyomunk, gondoljunk arra, hogy inkább egy függvényt kéne ebből a kódrészletből készíteni

Példa a DRY megsértésére

```
class Producer {
  private Queue queue;
  public void Produce() {
    lock (queue) {
      string item = "hello";
      queue.Enqueue(item);
    }
  }
}

class Consumer {
  private Queue queue;
  public void Consume() {
    lock (queue) {
      string item = queue.Dequeue();
    }
  }
}
```

DRY megoldás: a Queue szálbiztossá tétele

```
class Queue {
  private object mutex = new object();
  private List<string> items = new List<string>();

  public void Enqueue(string item) {
    lock (mutex) {
      items.Add(item);
    }
  }

  public string Dequeue() {
    lock (mutex) {
      string result = items[0];
      items.RemoveAt(0);
      return result;
    }
  }
}
```

DRY megoldás: a Queue szálbiztossá tétele

```
class Producer {
  private Queue queue;
  public void Produce() {
    string item = "hello";
    queue.Enqueue(item);
  }
}

class Consumer {
  private Queue queue;
  public void Consume() {
    string item = queue.Dequeue();
  }
}
```

Single Choice Principle (SCP)

- Bármikor arra van szükség, hogy a rendszer alternatívákat támogasson, akkor ideális esetben egyetlen helyen koncentrálódjon az alternatívák kezelése
- Ez a DRY és az OCP következménye
- Aka.: Single Point Control (SPC)
  - az alternatívák teljes listája egyetlen helyen koncentrálódjon

Tell, don’t ask (TDA)

- A metódusokat úgy hívjuk meg, hogy előtte nem vizsgáljuk a hívott objektum állapotát vagy típusát
- Az állapotot ellenőrző kódnak a célobjektumhoz kéne tartoznia, ez az ő felelőssége lenne

Példa a TDA megsértésére

```
class Pacman {
  void Step() {
    Field next = field.GetNext();
    if (next.IsFree) {
      next.Accept(this);
    } else {
      Thing other = next.GetThing();
      other.Collide(this);
    }
  }
}
```

TDA megoldás

```
class Pacman {
  void Step() {
    Field next = field.GetNext();
    field.Remove(this);
    next.Accept(this);
  }
}

class Field {
  void Accept(Thing t) {
    if (this.thing != null) {
      this.thing.Collide(t);
    } else {
      this.thing = t;
    }
  }
}
```

TDA

- A TDA megsértése a DRY megsértését is jelenti
- Sok problémát tud okozni:
  - a feltétel ellenőrzése lemaradhat néhány helyen
  - konkurencia problémák
- Hagyjuk a feltételek ellenőrzését a hívott objektumra
- A TDA megsértése azt jelenti, hogy a felelősségek rossz helyen vannak

Demeter-törvény (Law of Demeter: LoD)

- „Ne állj szóba idegenekkel!”
- Egy objektum függvénye csak az alábbi objektumok függvényeit hívhatja:
  - saját objektum
  - bejövő paraméterek
  - saját objektum adattagjai
  - az általa létrehozott objektumok
- Egy metódus ne hívja egyéb más objektumok tagjait

Példa a LoD megsértésére:

Vagy:

```
this.field.GetNext().GetThing().Collide(this);
```

Vagy:

```
Field next = this.field.GetNext();
Thing thing = next.GetThing();
thing.Collide(this);
```

LoD lehetséges megoldások

```
//Elfogadható:
Field next = this.field.GetNext();
next.Accept(this);


//Inkább:
this.field.MoveThingToNextField();
```

LoD

- Metódushívások láncolása esetén a lánc összes elemétől függünk
- Megoldás: delegálás
  - a lánc minden objektumába tegyünk egy függvényt, ami a lánc maradékát végrehajtja
- De vigyázzunk, hogy ez ne járjon a metódusok kombinatorikus robbanásával
  - ha mégis, akkor tervezzük át a dolgokat
- Csak az ismerős objektumokhoz delegáljuk a hívásokat!
- Így ha egy elem változik a láncban, az valószínűleg nem lesz hatással a lánc elejére

Összefoglalás

- OO fogalmak
- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Don’t Repeat Yourself (DRY)
- Single Choice Principle (SCP)
- Tell, Don’t Ask (TDA)
- Law of Demeter (LoD)