



# Szoftvertechnológia

# MODUL 1

**Mi a Szoftvertechnológia?**

**Hagyományos termékek vs szoftvertermékek**

**Szekvenciális életciklus modellek**

**Iteratív életciklus modellek**

# Köszönetnyilvánítás

- Ez a tananyag nem jöhetett volna létre az alábbi személyek elmúlt 30 évben tanúsított, lelkiismeretes munkája nélkül:
  - **Dr. habil Tick József** (életciklus modellek, UML)
  - **Szabó-Resch Miklós Zsolt** (tervezési minták, GIT)
  - **Dr. Erdélyi Krisztina, Nagy Tibor István** (UML gyakorlati alkalmazása)
  - **Sipos Miklós László** (agilis módszertanok)

# Miről szól ez a tantárgy?

- Hogyan kell szoftvert fejleszteni profin
- Milyen szinten vagyunk most? (SZTF1, SZTF2, HFT után és GUI-val párhuzamosan)
  - Profin tudunk full stack alkalmazást írni
  - A C# tudásunk megvan, a „tanfolyami” tematikákon végigmentünk
  - Nincs már hátra semmi jelentős dolog, keretrendszeret érdemes tanulni még
    - **Backend:** ASP.NET MVC keretrendszer → kötvál
    - **Frontend:** JS keretrendszer (pl. React, Angular, Vue.JS) → lesz kötvál (Angular)
    - **Mobil:** Cross platform keretrendszer (pl. Xamarin, Flutter, Ionic)
- Szinte teljes a palettánk...mi hiányzik???
- Mit tud egy 14 hetes előadás adni még nekünk???

A SZOFTVER MÉRETÉVEL MINIMUM NÉGYZETESEN  
NÖVEKSZIK ANNAK KOMPLEXITÁSA

# És ez miért baj?

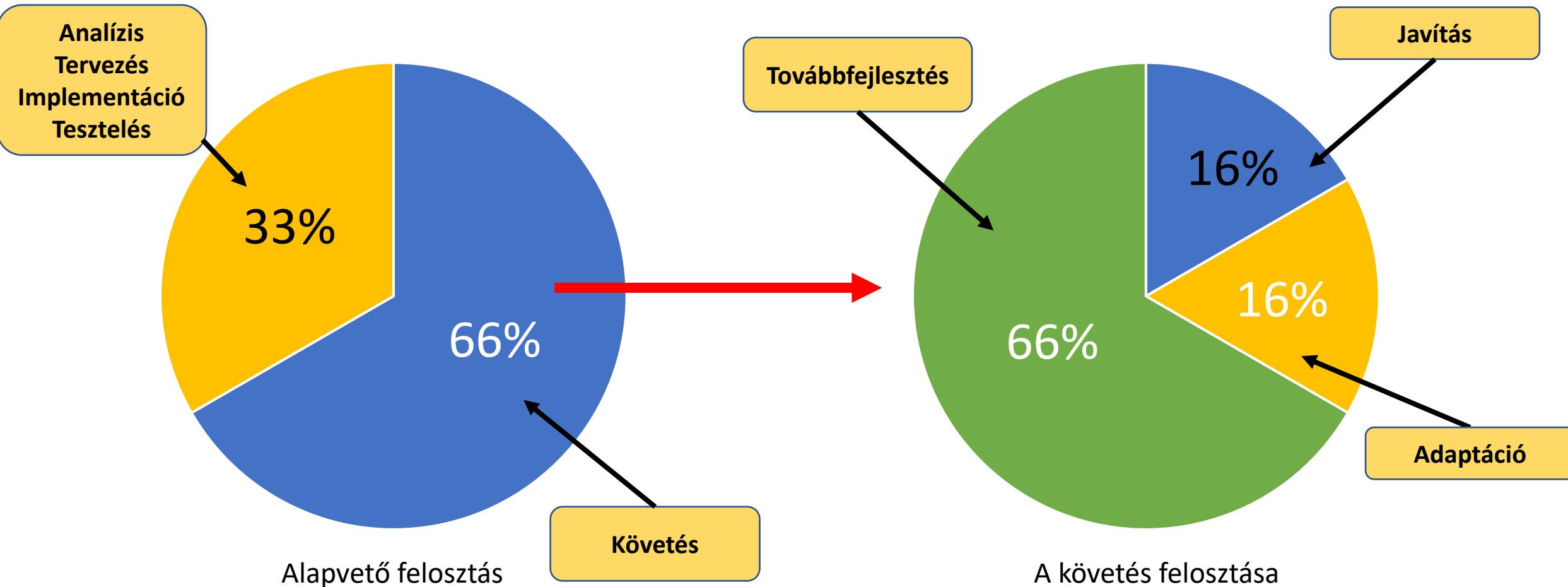
- Mire vagyunk képesek?
  - Egy 3 entitásos CRUD API alkalmazást 4 nap alatt lefejleszteni → előző félév (ti mondtátok)
  - Egy átlagos mini számlázó és raktározó alkalmazás minimum 20-30 tábla
    - **User management:** 3 tábla (users, roles, user+roles)
    - **Hozzáférés kezelés:** 2 tábla (resources, resources+roles)
    - **Telemetria (naplózás):** 5 tábla
    - **Valódi üzleti entitás táblák:** 8-10 tábla (lásd: VIR tantárgy → számlák, beszállító levelek, partnerek)
- Tehát méretben egy kis szoftver implementálása  $16 \times 4$  nap = 13 munkahét
- Vevő: „Természetesen már egy évvel ezelőttre kellett volna, csak az előző cég [...]”

**SOHA de SOHA** nem fogunk olyat hallani, hogy **RÁÉR**

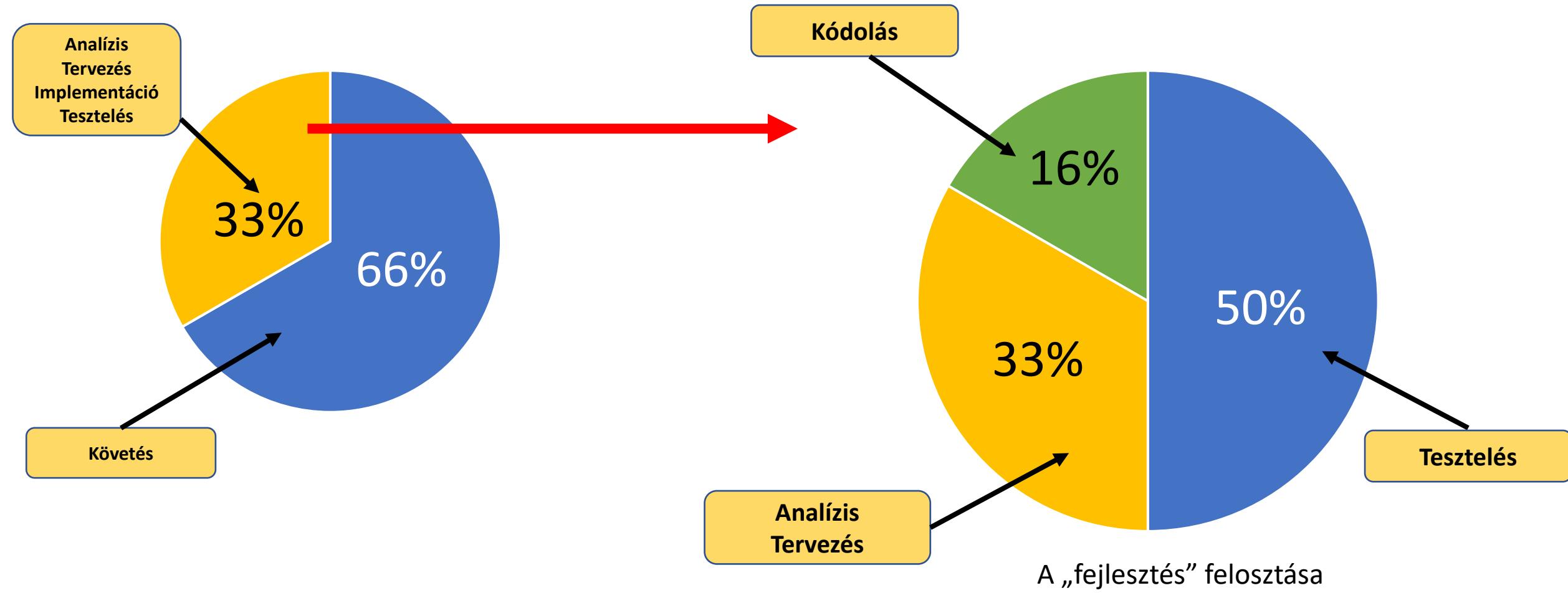
# Szakaszok költségarányai

7

- A világban megrendelt szoftverfejlesztési projektek költségarányai (vehetjük ezt időnek is)



# A fejlesztés költségarányai



- **Analízis**

- A probléma megértése a megrendelő elmondása alapján
- Specifikáció megfogalmazása

- **Tervezés**

- Specifikacióból adatbázis terv, rendszerterv, OOP osztályok tervezése, rétegek kialakítása, stb.

- **Implementáció**

- A tervezés alapján a szoftver lekódolása

- **Tesztelés**

- Az elkészült szoftver letesztelése → ezt már tudjuk, hogy nem utólag kéne

- **Javítás**

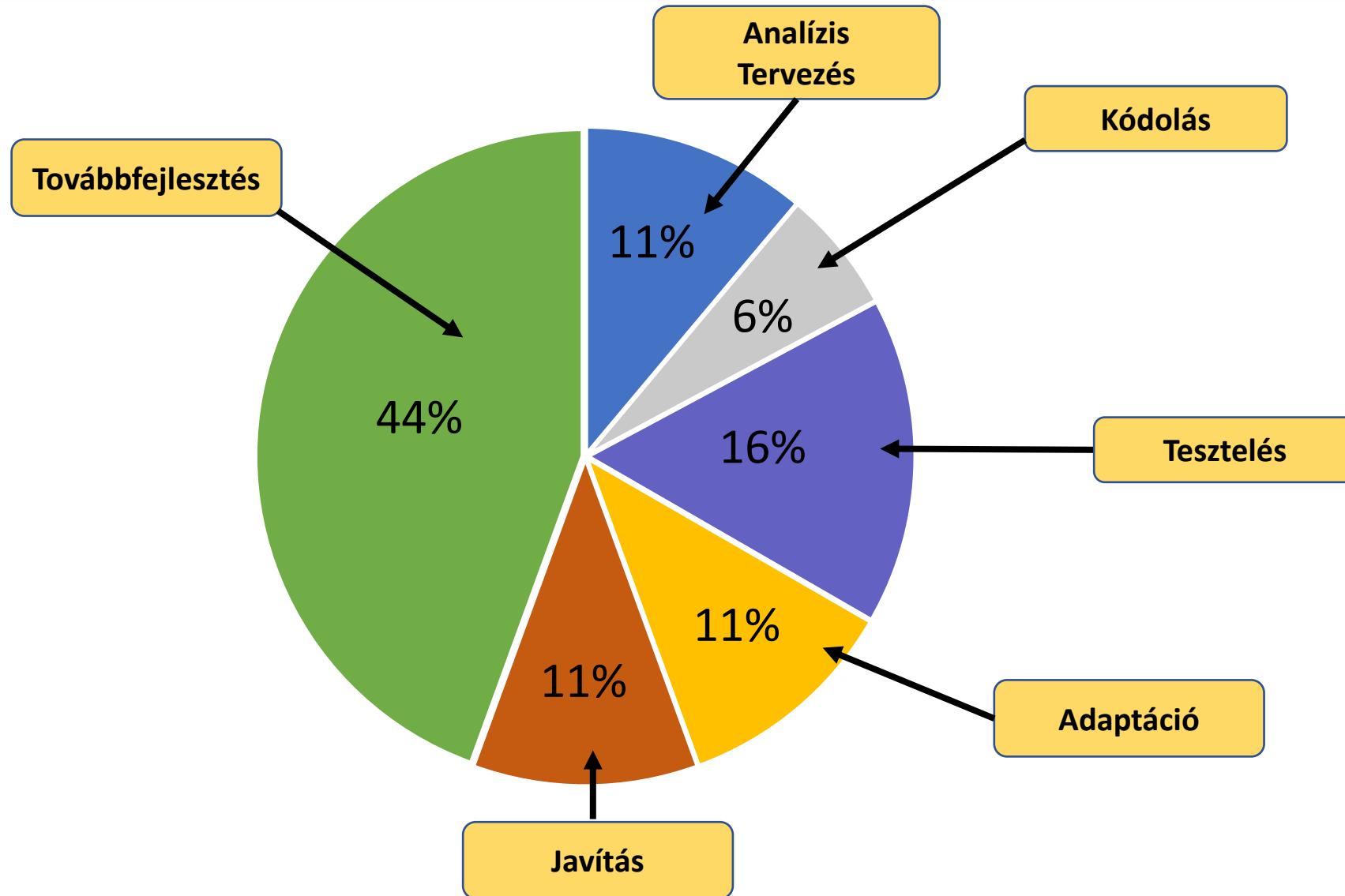
- Fejlesztői részről minden tökéletesen működik, de kibuknak hibák (általában rossz spec. miatt)

- **Adaptáció**

- Megrendelő adottságaihoz alakítás (pl. neki van már szervere, levelezése, adatbázisa)

# Egy grafikonon minden

10



- HFT féléves feladat
  - Kódolás: volt
  - Tesztelés: érintőleg
  - Tervezés: picit
- minden más nem volt a része
  - Analízis: nem volt
  - Továbbfejlesztés: nem
  - Javítás: nem volt
  - Adaptáció: nem volt
- **HFT: 18%-át fedtük le**

# Akkor meddig is tart?

- Mini raktárkezelő példa: 13 munkahét egy főnek
- De HFT tapasztalat alapján mondtuk ezt → tehát ez csak a munka 18%-a
- Valójában kb. **361 munkanap** a megkereséstől a vevőtől való teljes megszabadulásig ☺
  
- Gondoljunk egy órabérre, legyen nettó 3500 ft / óra
- Szoftver ára: 10.000.000 ft + áfa
- Elkészülési idő: 19 hónap → 1 év 7 hónap

Ezt senki sem fogja kifizetni vagy kivárni

- Tippünk: **10M Ft** és **19 hónap**
- Reálisan erre maximum **3M Ft**-ja lesz egy magyar vevőnek és körülbelül **6 hónapja**
- Ha csak nem talál a piacon egy olyan szoftvert, ami amúgy teljesen megoldja ezt neki
  - Ami ráadásul ingyenes
  - Vagy legalábbis havidíjas (MS Office kb. 5 euró havonta...)
- És lehetséges ezt megugrani, nagyon is!

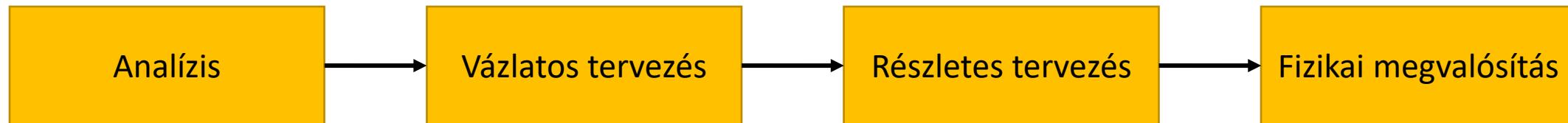
- Kizárolag csapatmunkában
- Multibranch GIT környezetben
- Projektmenedzsment tool-ok támogatásával
- Újrahasznosított komponensekkel
- Frameworkök használatával
- Folyamatos teszteléssel
- Folyamatos integrációval
- Jól bevált, profik által javasolt kódmintákkal
- Folyamatos vevői véleményeztetéssel
- Fenntartható, moduláris kód írásával

Ezt fogjuk megtanulni, ezt jelenti a Szoftvertechnológia

- 1960-as évek szoftverfejlesztése
  - 2% azonnal működött
  - 3% javítás után működött
  - 20% alapos átdolgozás után ment
  - 30% soha nem ment, de kifizették
  - 45% soha nem ment rendesen
- A kor jellemzői
  - Gyenge hardver
  - Fejlesztői eszközök nem voltak → text editor + compiler
  - Monolitikus programozás (spaghetti kód) → SZTF1 féléves feladatotok pont ilyen ☺
  - Team munkához semmilyen eszköz nincsen (manual merge)
  - Konfiguráció változást nem támogatja a szoftver
- Megoldás: új paradigmák (pl. OOP), modularizálhatóság megjelenése

# Hagyományos termék előállítása

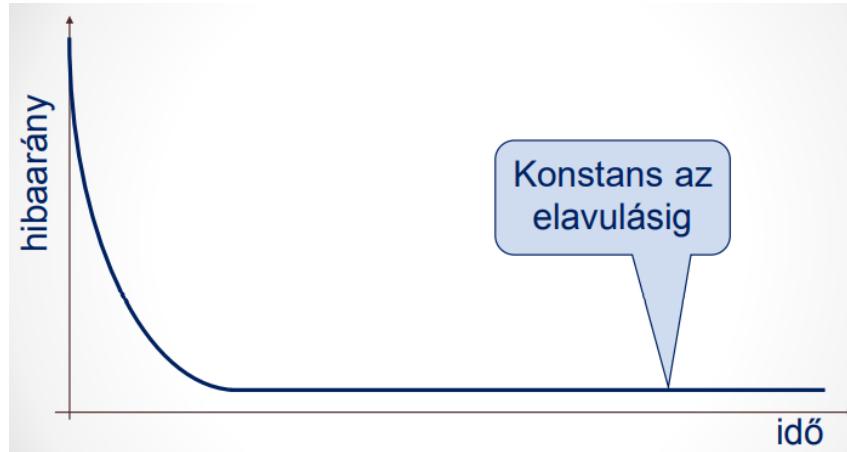
15



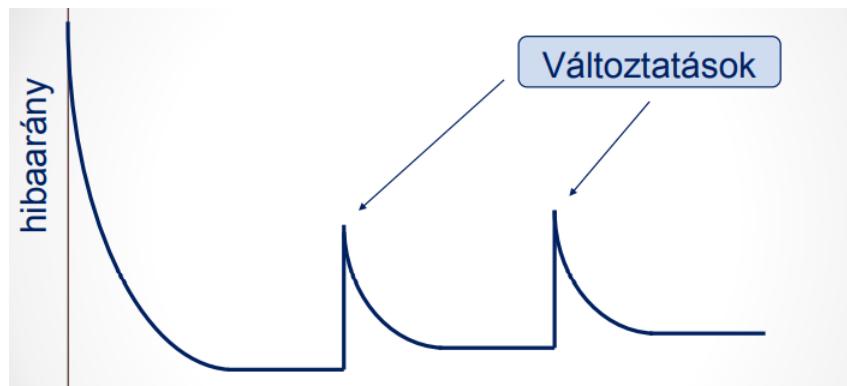
- Gyerekbetegségek:
  - Tervezási, gyártási hibák
  - Hibás alapanyagok
- Elkopás okai:
  - Öregedés, szerkezeti elváltozások

# Szoftver termékek hibaarány görbéje

16



- Nincsen öregedés, kopás



- De van helyette állandó módosítási igény
- Az újabb funkciók, újabb hibákat eredményeznek
- Ezek a hibák összeadódnak



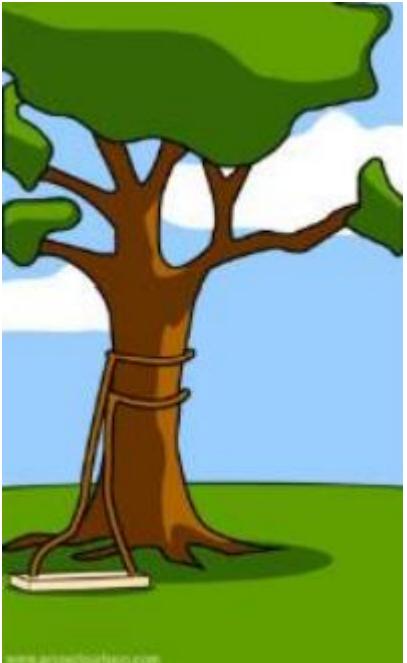
Amit a megrendelő  
elmondott



Amit a projektvezető  
megértett belőle



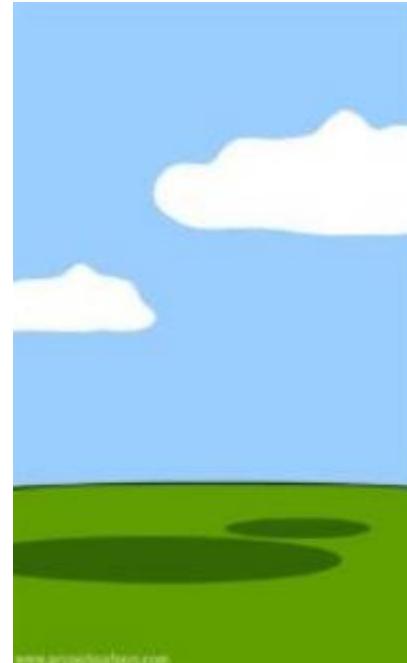
Amit a tervező  
megtervezett



Amit a programozó  
leprogramozott



Ahogy a marketinges  
prezentálta



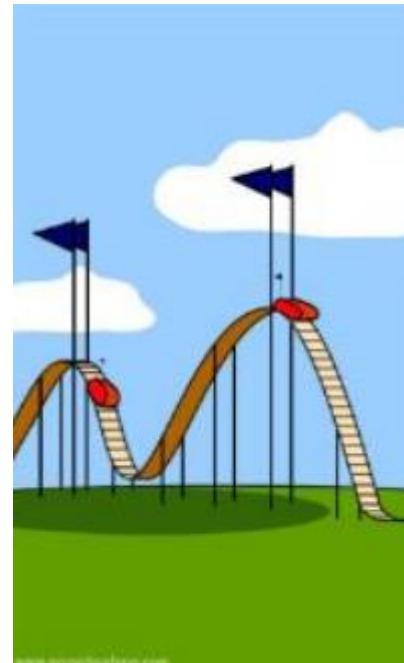
Ahogy a terméket  
dokumentálták



Amit a megrendelőnek  
átadtak



A megrendelő  
elképzése az árról



Amit a megrendelőnek  
kiszámláztak



A bemutatási effektus



Ahogy a rendszer a  
terhelést bírta



Amire a megrendelőnek  
igazából szüksége lett volna

- **Szekvenciális modellek**

- Vízesés modell
- V-modell

Céljuk a szoftverfejlesztés lépéseinak meghatározása

- **Iteratív/inkrementális modellek**

- Evolúciós modell
- Formális rendszerfejlesztés
- Újrafelhasználás-orientált fejlesztés
- Inkrementális fejlesztés
- RAD modell
- Spirál modell

- **Agilis modellek**

- XP, Scrum, Kanban

- **Rational Unified Process (RUP)**

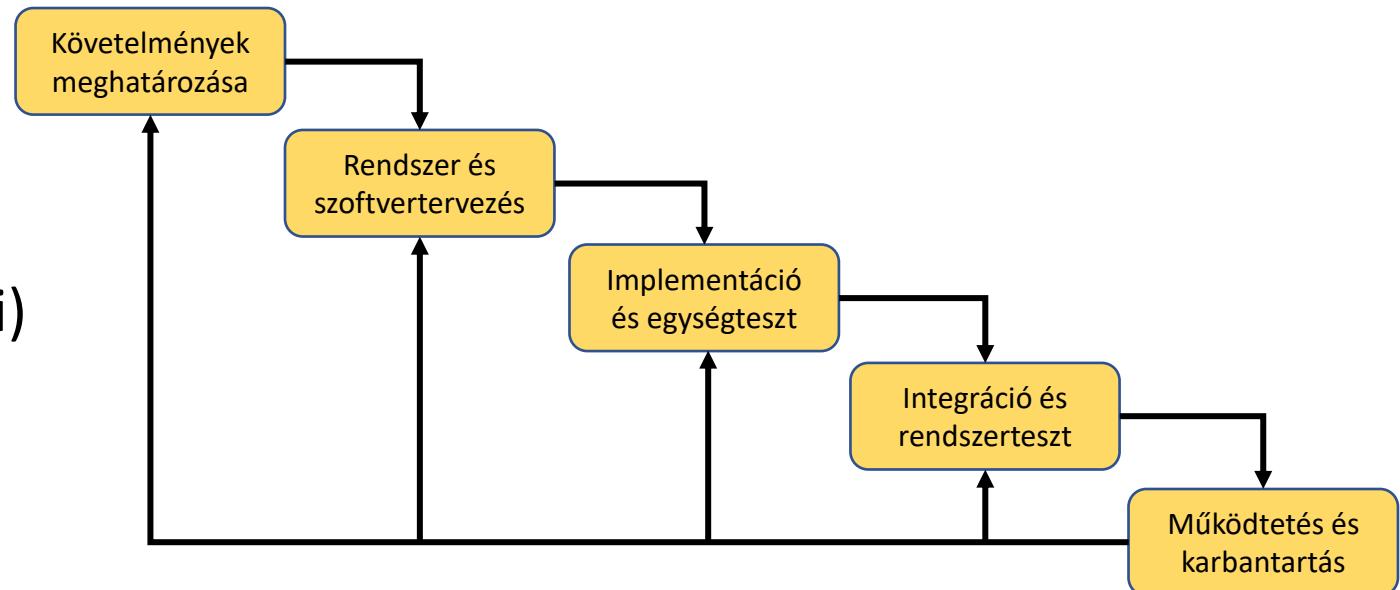
# Vízesés modell

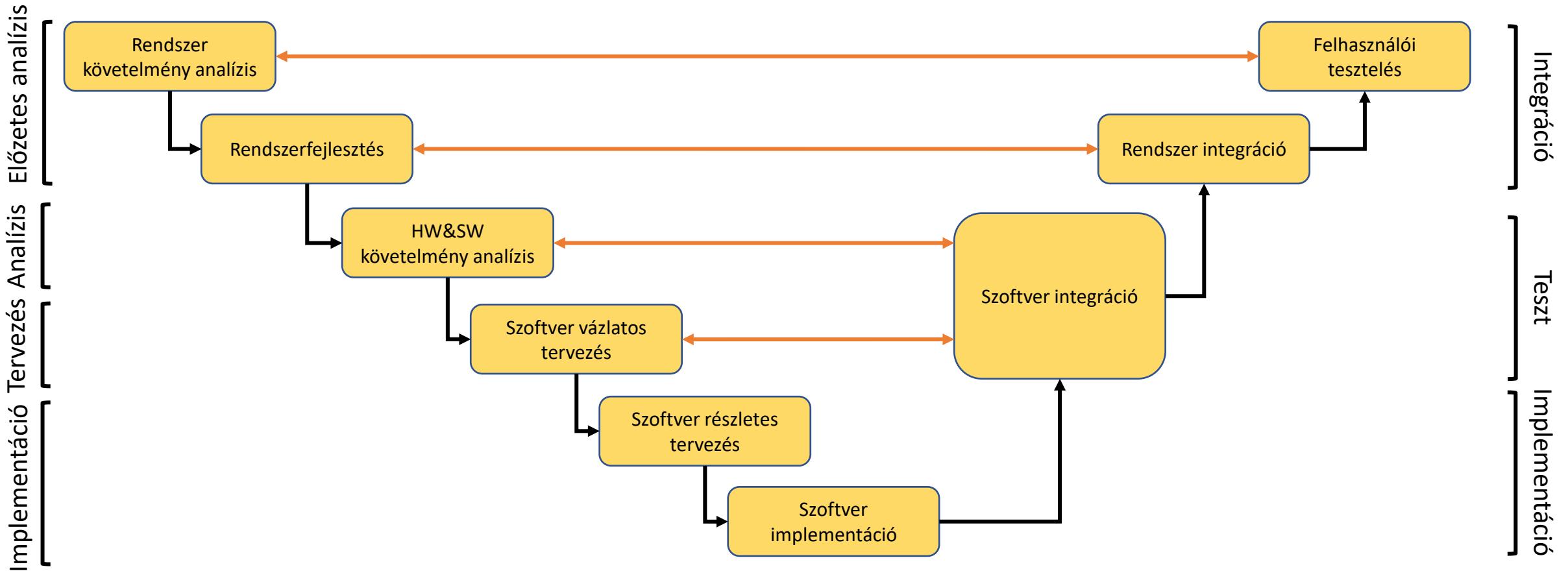
22

- 1970: W. W. Royce
- Hagyományos mérnöki szemlélet követése
- Leginkább elterjedt modell (mert régi)

## Problémái:

- Valós projektek nem így működnek
- minden a specifikáció minőségétől függ
- Nagyon későn lát a megrendelő működő programot
- Kezdeti bizonytalanságöt nehezen kezeli
- Tesztelés szerepe nem elégéhangsúlyos

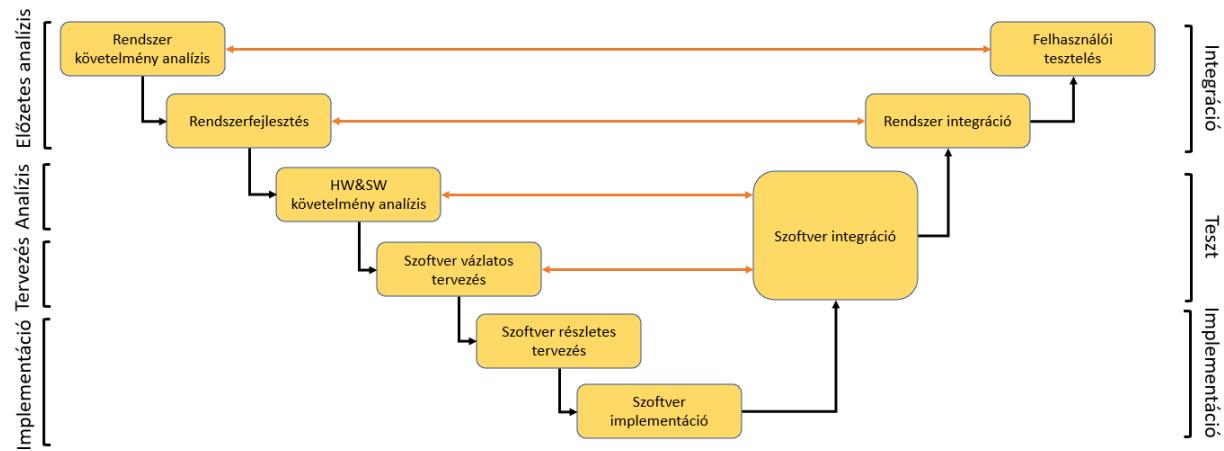




# V-modell

24

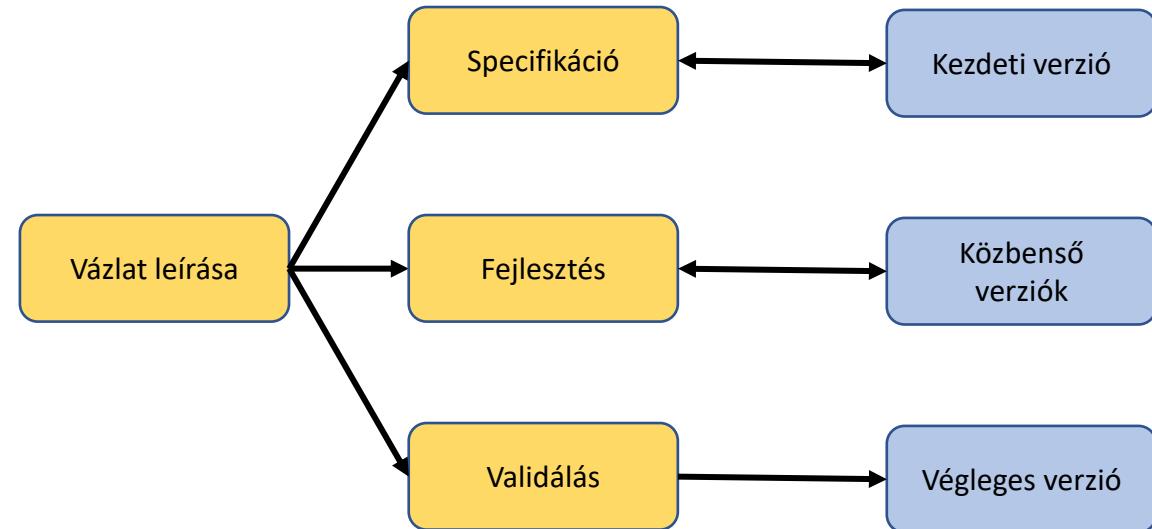
- 1991: német védelmi minisztérium
- Vízesés modell továbbfejlesztése a tesztelés hangsúlyossá tételevel
- Szigorú dokumentálás
- Nem igazán küszöböli ki a vízesés modell problémáit
- Kritikus rendszerek fejlesztésénél igen előnyös



# Evolúciós modell

25

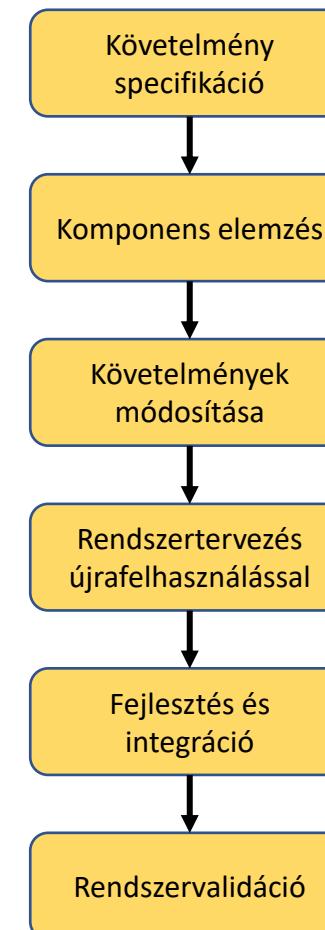
- Kifejlesztünk egy kezdeti implementációt
- Véleményeztetjük a felhasználókkal
- Sok verzión keresztül finomítjuk, amíg el nem érjük a kívánt rendszert
- **Problémák:**
  - Nehezen menedzselhető a folyamat
  - Minőségbiztosítási problémák
  - Speciális eszközök és technikák igénye



# Újrafelhasználás-orientált fejlesztés

26

- A fejlesztési idő nagy mértékben lerövidíthető
- Lényegesen olcsóbbá válik a fejlesztés
- A rendszer minősége és megbízhatósága jobb, mert ellenőrzött komponenseket építünk be
- Nagy library gyűjtemény esetén nehéz a pontos komponens kiválasztása

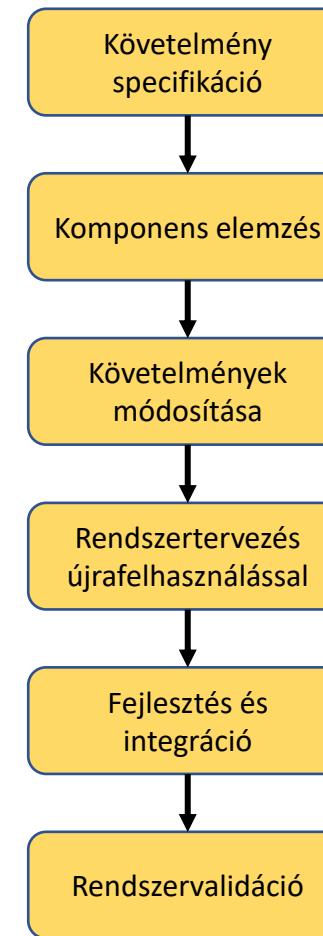


# Újrafelhasználás-orientált fejlesztés

27

- C# megvalósítás példa
  - Az első projektünkben megírjuk a **user** és **role** kezelés tábláit, **repóját** és **logic rétegét**
  - A **user** és **role** entitásokat később **származtatással** bővíthetjük extra tulajdonságokkal
  - Készítünk **generikus repository-kat**, **logicokat** (spec. eltérés?)
  - Ezekből **DLL-eket** készítünk
  - Készítünk saját projekt templateket, ahol ezek már alkalmazva vannak
  - A DLL-eknek saját **nuget repository-t** gyártunk
  - A már legyártott éles szoftverek is frissíthetik magukat a saját **nuget repositoryból** (utólagos javítások automatikusan bekerülnek)
- CMS rendszerek is jó példák erre (pl. Wordpress)

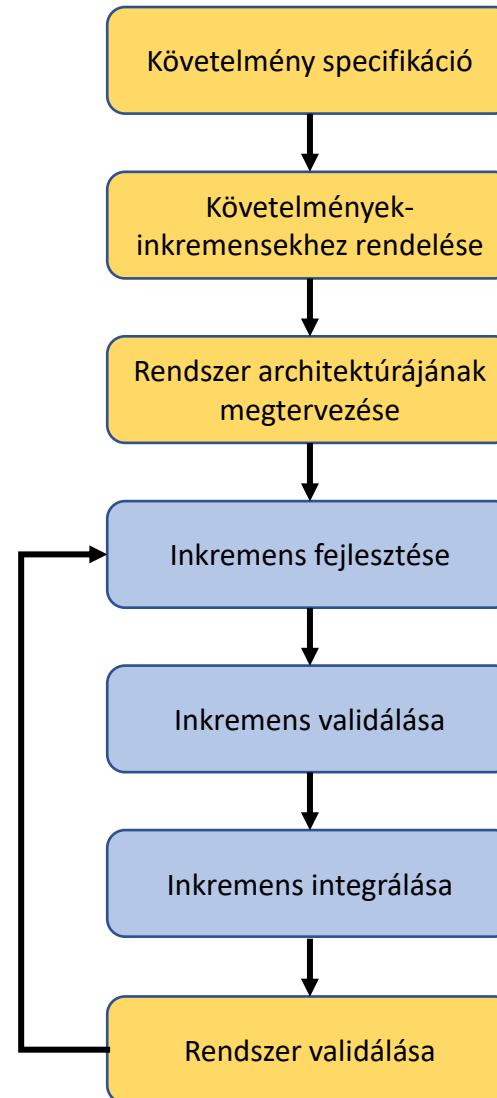
Semmi üzleti/emberi/mérnöki értéke nincs annak, hogy egy szoftvert mindenkoról írunk.



# Inkrementális fejlesztés

28

- A rendszert kisebb egységekre bontjuk
- minden egység önálló fejlesztés – validálás – integrálás folyamatot kap
- minden inkremens külön egyeztetett, elkészülte után a megrendelő használatba veheti
- Cél a komplexitás csökkentése
- Megrendelőnek lehetősége van bizonyos követelményekkel kapcsolatos döntések elnapolására
- Mi lehet inkremens?
  - Tipikusan alrendszer!



- Konkrét példa: OE-NIK hallgatói fórum specifikáció
  - Lehessen kérdéseket felenni, válaszokat adni
  - Lehessen témákat indítani
  - A témaik is legyenek nagyobb témaikba gyűjthetők
  - Legyen pontrendszer a hasznos válaszokért
  - Legyen gamifikáció → lehessen fejleszteni az avatarunkat a pontokból
  - Lehessen hirdetni az appban
  - Lehessen tanárokknak is regisztrálni
  - Lehessen a tanárok elől topicokat elrejteni
  - Lehessen képeket és videókat feltölteni
  - Lehessen fb, google, linkedin fiókkal 1 kattintással belépni

- Konkrét példa: OE-NIK hallgatói fórum specifikáció

- **Core rendszer**

- Lehessen kérdéseket felenni, válaszokat adni
    - Lehessen témákat indítani
    - A témák is legyenek nagyobb témákba gyűjthetők
    - Lehessen képeket és videókat feltölteni
    - Lehessen fb, google, linkedin fiókkal 1 kattintással belépni

I.

A Core fejlesztés tipikusan újrafelhasználás-orientáltan megoldható (CRUD alkalmazás)

II.

- Legyen pontrendszer a hasznos válaszokért
    - Legyen gamifikáció → lehessen fejleszteni az avatarunkat a pontokból

III.

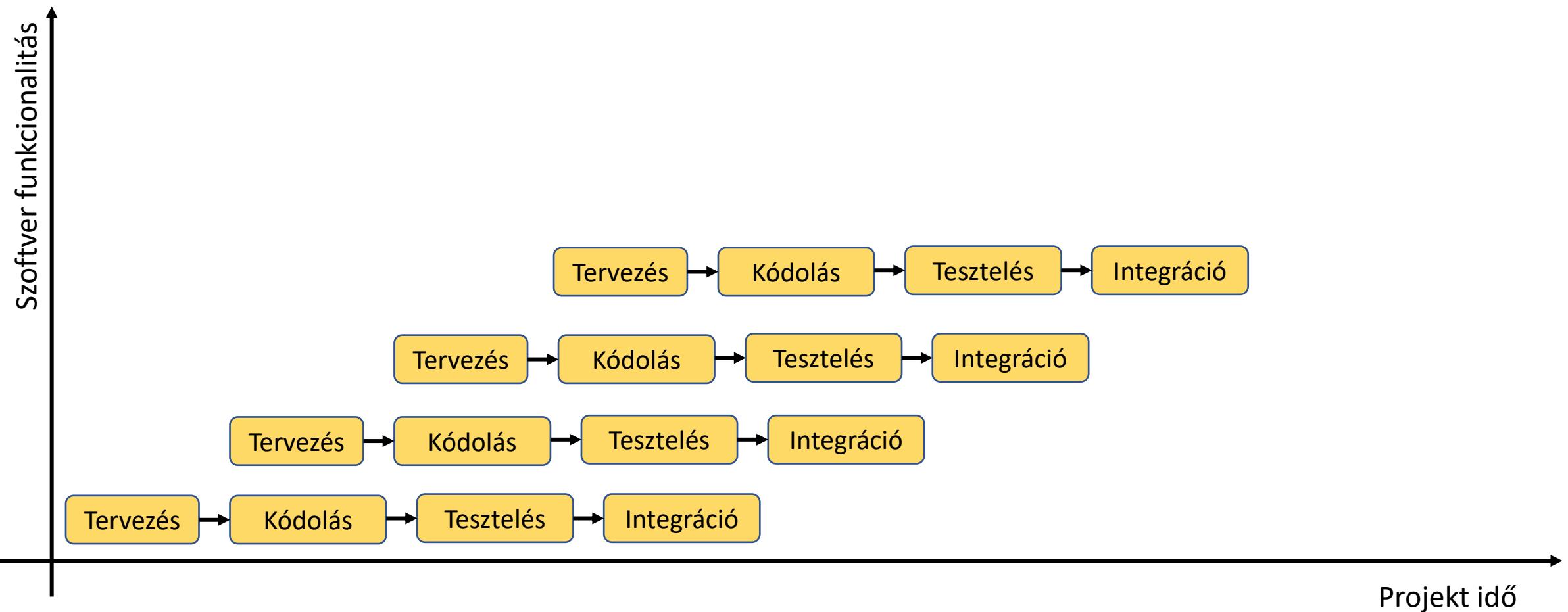
- Lehessen hirdetni az appban

IV.

- Lehessen tanárokknak is regisztrálni
    - Lehessen a tanárok elől topicokat elrejteni

# Inkrementális fejlesztés

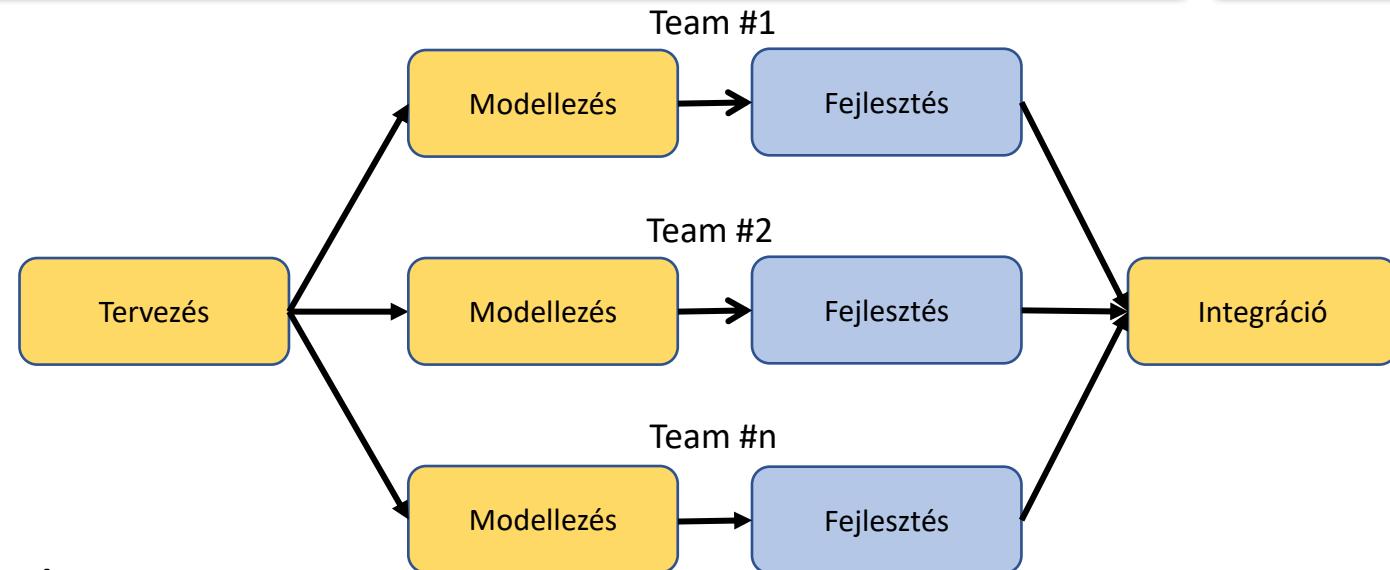
- Lehetővé válik pipeline-alapú fejlesztés!



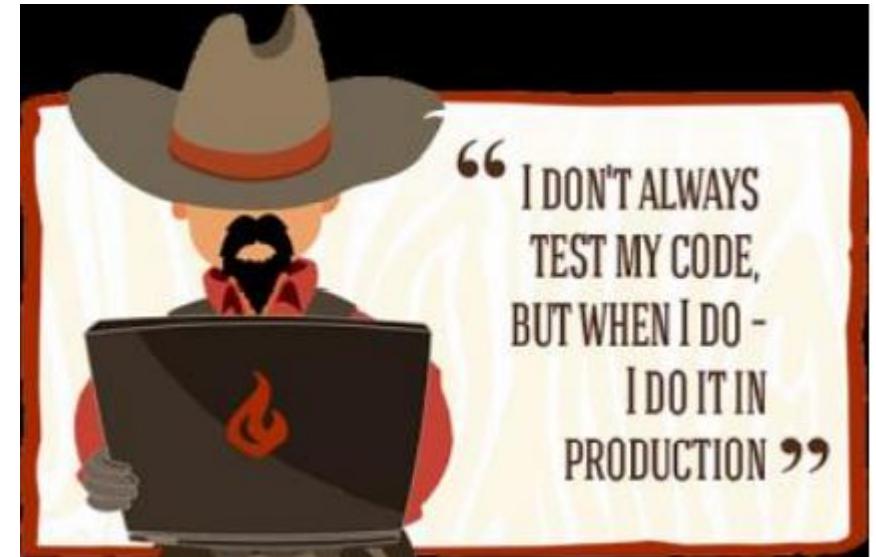
- **Előnyei**

- A szoftver már menetközben használhatóvá válik a megrendelő számára
- Kritikus követelmények teljesülnek először
- Kisebb a kockázata a projekt kudarcának, biztonságos fejlesztési koncepció
- A legkritikusabb funkciókat teszteljük legtöbbet

- Rapid Application Development
- Vízesés modell high-speed adaptációja
- **Építőelemei:**
  - Újrafelhasználás-orientált fejlesztés
  - Komponensekre bontás
  - Kódgenerálás
- Sokoldalú, párhuzamosan dolgozó teamek
- Megrendelő szakemberei is a teamek részei lehetnek
- **Hátrányai:**
  - Nagy projekt → hatalmas humán erőforrás igény
  - Ha nehezen modularizálható a rendszer, akkor nem működik



- A kódírás előtérbe helyezése az összes többi fázissal szemben
- **Előnyök**
  - Nagy szabadságfok a megoldásnál
  - Válságkezelést gyorsan orvosolja
  - Nem köti az architektruális terv a programozót
- **Hátrányok**
  - Team-munka lehetetlen
  - Nincs minőségbiztosítás
  - Nem fenntartható a kód
  - Gyenge dokumentáció
  - Nagy projekt esetén nem használható
  - Nem lehet vele jól keresni 😞



Néha érdekes lehet ilyen egyszemélyes projektet is bevállalni, jót tesz a kreativitásnak.

# Köszönöm a figyelmet!

Kérdés esetén e-mailben szívesen állok rendelkezésre.



# Szoftvertechnológia



ÓBUDAI EGYETEM  
NEUMANN JÁNOS INFORMATIKAI KAR

# MODUL 2

**Mi a gond az eddigi modellekkel?**

**Agilis elvek**

**Extreme Programming**

**Scrum**

# Mi a baj az eddigi modellekkel?

- Mindegyik erősen függ a specifikáció minőségétől
- A megrendelő későn lát működő szoftvert
- Még mindig nincs közös nyelv
- Nehéz felelőst találni, hogyha nem az készül el, amire a vevő gondolt
- A fejlesztők nem igazán szeretik a dokumentáció készítést

# Manifesto for Agile Software Development

4

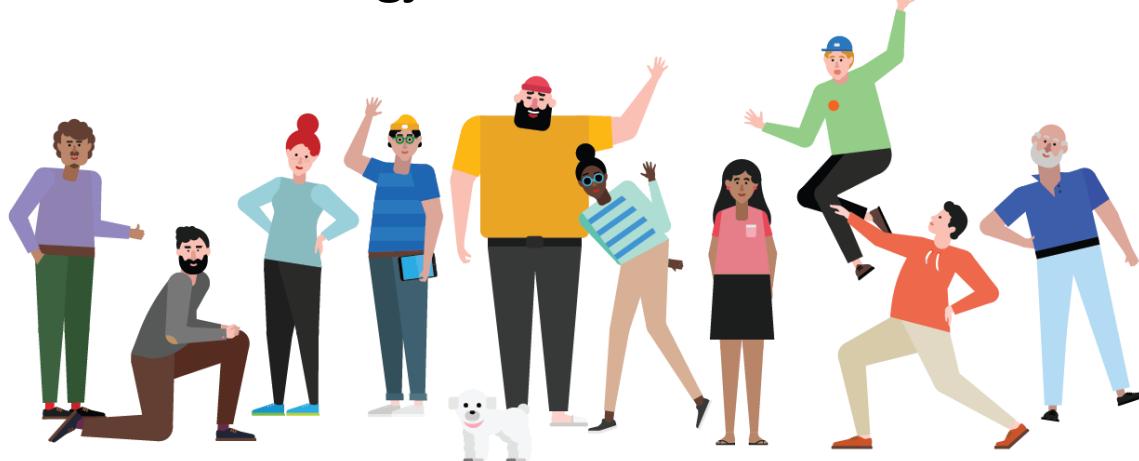
- 2001: 17 neves szoftverfejlesztő megalapította az Agilis Szövetséget és megfogalmazták a kiáltványukat az agilis szoftverfejlesztésért
- Főbb pontok
  - **Egyének és személyes kommunikáció preferálása**
    - Módszertanokkal és eszközökkel szemben
  - **Működő szoftver preferálása**
    - Átfogó dokumentációval szemben
  - **A megrendelővel történő együttműködés preferálása**
    - A szerződéses egyeztetéssel szemben
  - **A változás iránti készség preferálása**
    - A tervezek szolgai követésével szemben



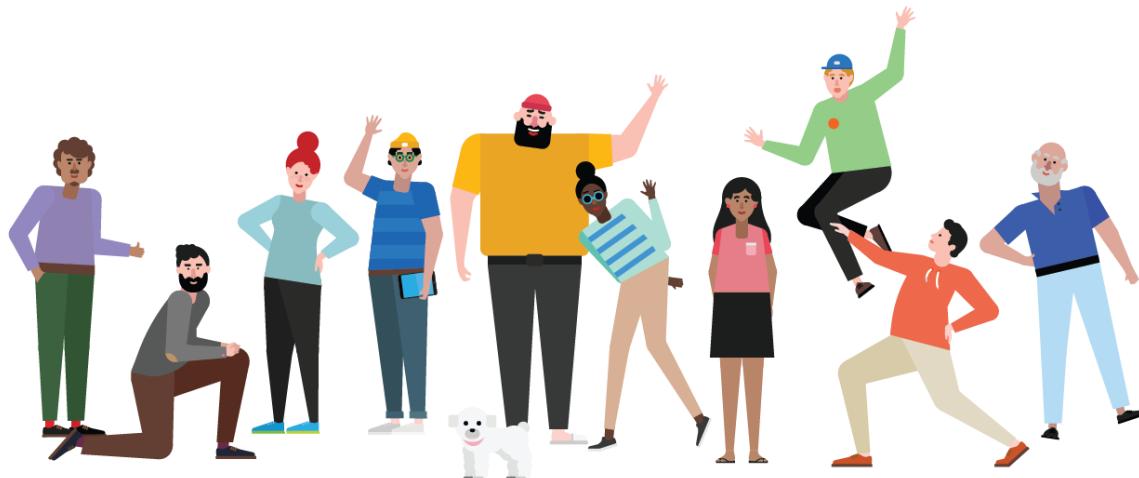
Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

# Agilis elvek

- Vevői elégedettség minél magasabb fokú elérése
- Flexibilitás a követelmények változásával szemben
- Működő inkremens minél gyorsabb átadása
- Üzleti szakemberek és fejlesztők legyenek napi kapcsolatban
- Motivált szakemberek gyűjtése a projektbe, feltételek biztosítása
- Szemtől-szembeni párbeszéd szorgalmazása az információcserére
- A haladás legjobb mértéke a működő szoftver

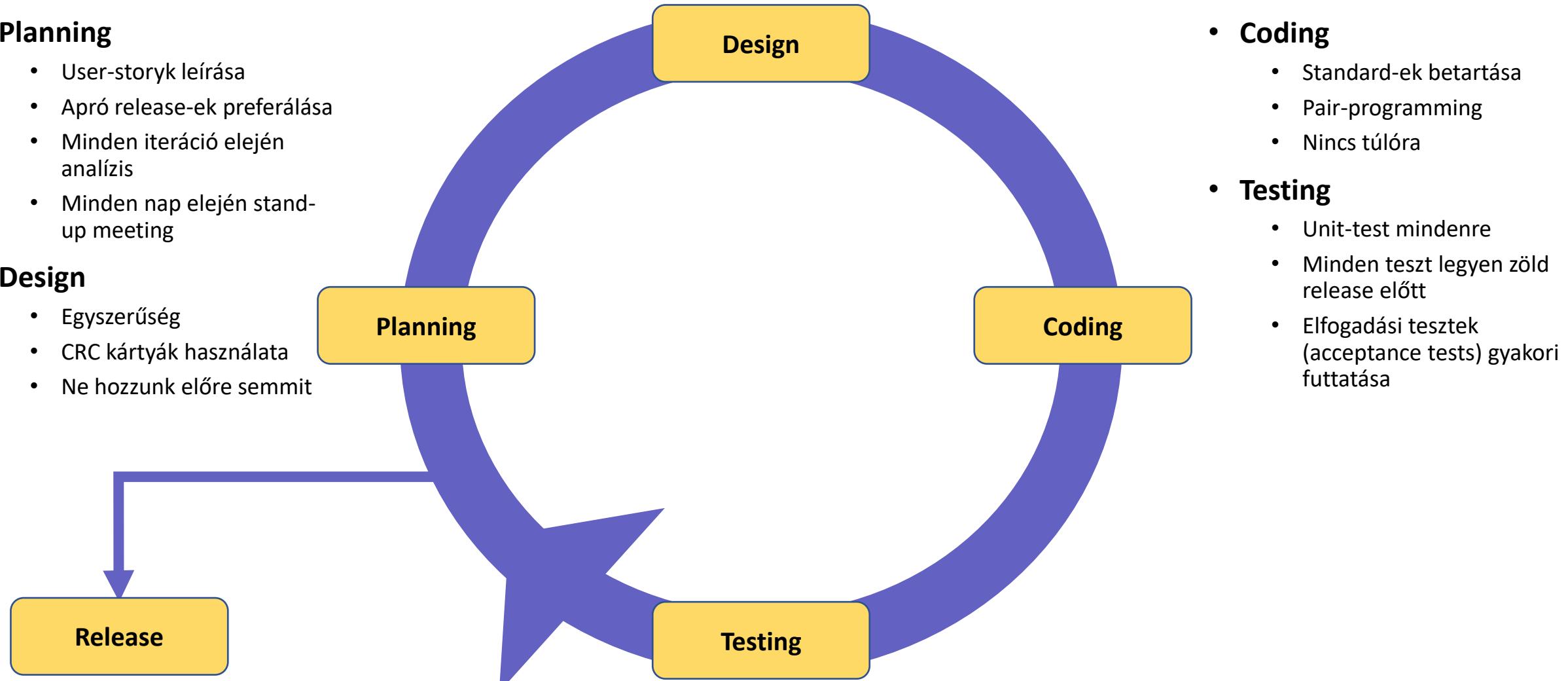


- Fenntartható fejlődés ösztönzése
- Fejlesztőknek és a megrendelőnek egy állandó ütemet kell fenntartani
- Jó tervre és a kiváló műszaki színvonalra ügyelni kell
- Csak az igazán fontos feladat elvégzése
- Önszerveződő, aktív teamek létrehozása
- Teameknek rendszeres önvizsgálat tartása



- 1999: Kent Beck
- Inkrementális fejlesztés + OOP + Agilis elvek
- 4 aktivitásból áll
  - **Planning** (analízis)
  - **Design** (tervezés)
  - **Coding** (implementáció)
  - **Testing** (tesztelés)
- Alapvető értékei
  - **Kommunikáció**: közös gondolkodás, aktív alkotói léggör
  - **Egyszerűség**: core funkciók először, apróságok utoljára (core first, shine last)
  - **Visszacsatolás**: tesztelők is, megrendelő is, fejlesztők is!
  - **Bátorság**: merjünk új technológiákhoz nyúlni
  - **Tisztelet**: egymás munkája iránt maximálisan

- **Planning**
  - User-storyk leírása
  - Apró release-ek preferálása
  - minden iteráció elején analízis
  - minden nap elején stand-up meeting
- **Design**
  - Egyszerűség
  - CRC kártyák használata
  - Ne hozunk előre semmit



- **Coding**
  - Standard-ek betartása
  - Pair-programming
  - Nincs túlóra
- **Testing**
  - Unit-test mindenre
  - minden teszt legyen zöld release előtt
  - Elfogadási tesztek (acceptance tests) gyakori futtatása

# XP előnyök és hátrányok

9

- **Előnyök**

- Jól kezeli a követelmények változását
- Költséghatékonyabb az eddigieknél
- Nagyobb megrendelői elégedettség
- Alacsony kockázati tényező
- Eddigi legjobb megoldások megjelennek benne

- **Hátrányok**

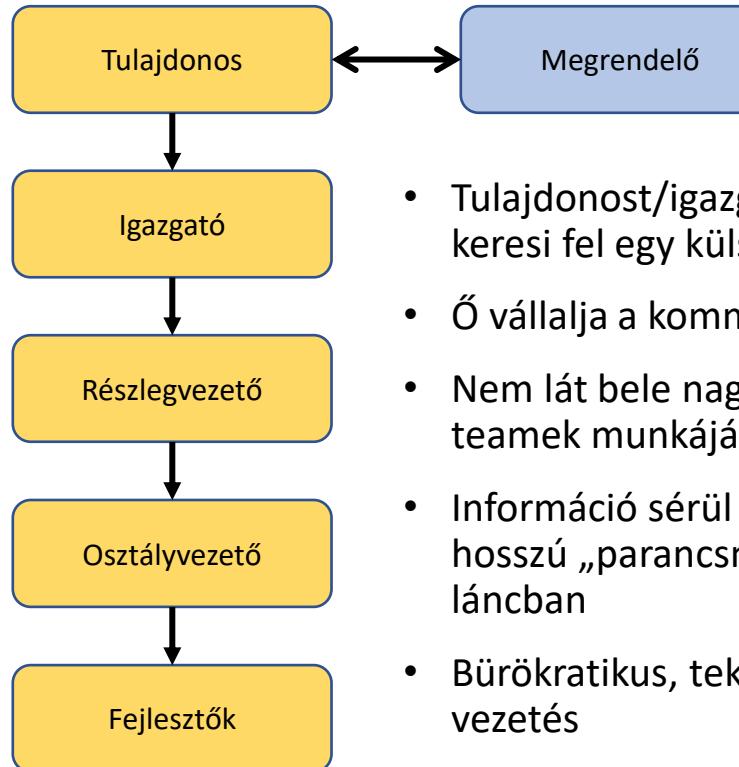
- Minőség erősen függ a megrendelő közreműködési szándékától
- Nem terv centralizált, hanem kód-centralizált
  - Nehéz az újrafelhasználása később
  - Nehéz később „elővenni”
- Nem fektet elég hangsúlyt a dokumentáció készítésére

- 1986: Hirotaka Takeuchi és Ikujiro Nonaka
- Sikeres vállalatok tanulmányozásából született a Scrum
- Eddigi módszerek
  - elméletek → gyakorlat
- Scrum
  - Jó gyakorlatok → rendszer
- Szerepkörök
  - **Disznók:** Scrum Master / Product Owner / Team
    - Szinte önfeláldozás az elvárt (disznó önmagát adja az ételbe)
  - **Csirkék:** Menedzserek / tulajdonosok / haszonélvezők
    - Ez is csak egy üzlet (csirke tojik egy tojást az ételbe)



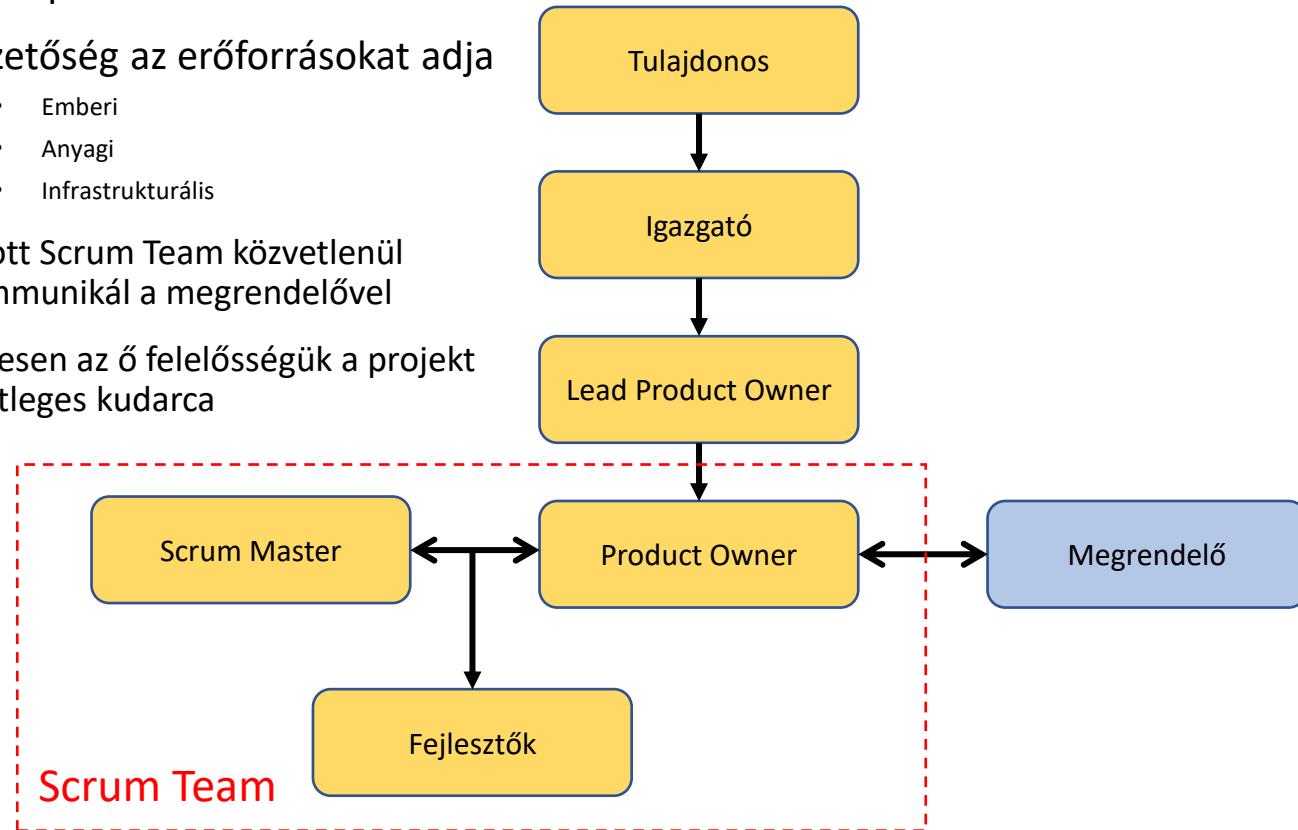
# Disznók? Csirkék?

11



- Tulajdonost/igazgatót keresi fel egy külső partner
- Ő vállalja a kommunikációt
- Nem lát bele nagyon a teamek munkájába
- Információ sérül ebben a hosszú „parancsnoki” láncban
- Bürokratikus, tekintélyelvű vezetés

- Scrum teamek, mint apró startupok
- Vezetőség az erőforrásokat adja
  - Emberi
  - Anyagi
  - Infrastrukturális
- Adott Scrum Team közvetlenül kommunikál a megrendelővel
- Teljesen az Ő felelősségeük a projekt esetleges kudarca



- **Product Owner**

- A csapat tagja, a saját cégünk embere, kollégánk
- De a **megrendelő érdekeit képviseli**
- Eljátssza, beleéli magát, hogy ő a megrendelő „ügynöke”
  - Tisztában van a megrendelő céljaival/problémáival
  - Tisztában van a prioritásokkal
- Megszervezi a rendszeres demókat
- Nem fejlesztő, nincs is feltétlenül ilyen múltja
- Érti a gazdasági folyamatokat, érti az IT-t, az architekturát
  - (Gazdaságinformatikus szak kiváló kezdet ide)
- Product Backlog kezelője



# Scrum szerepkörök

13

- **Scrum Master**

- A csapat tagja, a saját cégünk embere, kollégánk
- Hagyományos projektmenedzser szereppel egyezik a feladata
- Felügyeli a folyamatokat, betartattja a Scrum szabályait
- Konfliktusokat kezel, akadályok elhárítását irányítja
- A meetingeket ő szervezi, ő vezeti
- A Scrum csapat feje



- **Scrum Team**

- 5-9 fő alkotja
- Színes csapat szükséges: elemző, fejlesztő, tesztelő, matematikus
- Ők végzik a tényleges fejlesztést
- Felelősségekük, hogy egy sprintre bevállalt feladatokat elvégezzék
- **Fejlesztői fokozatok**
  - Elvileg mindenki egyenlő egy Scrum teamen belül
  - De mindenhol van junior → medior → senior → architect szint megkülönböztetés

- **Tesztelő**

- Hivatalosan mindenki a saját kódjának a tesztelője (unit testek)
- Unit tesztek után peer review más fejlesztővel
- De mindenhol van manuális teszt, amit nem a fejlesztők végeznek

- **User Story, Task**

- Egy specifikációból eredő feladat (pl: food management)
- Taskokra bomlik szét, ezeket veszik magukra a fejlesztők

- **Sprint**

- 1-4 hét hosszú fejlesztési szakasz (általában 3 hét)
- Addig jönnek újabb és újabb sprintek, amíg a Product Backlogból el nem tűnnek a User Story-k
- A sprint vége egy leszállítható szoftver

- **Product Backlog**

- Még elkészítésre váró Story-k gyűjtőhelye
- PO tartja karban (prioritásokat rendel a story-khoz)
  - **PO:** üzleti érték becslése
  - **Team:** ráfordítás becslése
- **ROI:** Return of Investment = üzleti érték / ráfordítás

- **Sprint Backlog**
  - A Product Backlog de csak az adott sprintre bevállalt storykra szűrve
- **Burn down/up chart**
  - Napi eredmény diagram
  - Megmutatja, hogy a csapat mennyire tartja az eredeti ütemtervet
- **Impediment**
  - Akadály, amely a munkát hátrálhatja
  - Tipikusan valamilyen munkahelyi probléma
  - Scrum Master feladata elhárítani

- **Sprint (pre-)grooming**
  - Architect(ek) + Scrum Master
  - Storyk ellenőrzése, pontozása (nehézség/idő → Fibonacci számokkal)
- **Sprint planning**
  - A Team bevállal storykat (product backlog → sprint backlog)
  - Figyelembe veszik a pontozást



- **Daily stand-up**

- minden nap ugyanakkor
- Egész team részt vesz
- Kb. 15 perc, állva, eszközök nélkül, szóban
- Mit csináltam tegnap? Mit fogok csinálni ma? Milyen problémám van?
- Pszichológiai fegyver a lusták ellen ☺

- **Sprint refinement**

- Hetente 1 maximum (3 hetes sprintben maximum 2 alkalom)
- Storyk/Taskok áttekintése
- Gyors hibaelhárítások
- Nem része a Scrumnak, de alkalmazzák

- **Egyéb meeting**

- Bárki bárkivel együtt dolgozik egy problémán, csapat részhalmaza egyeztet

- **Sprint review**

- Team + Scrum Master + Product Owner
- „User acceptance test”
- Eredmények bemutatása, PO dönti el, hogy sikeres-e a sprint
- Kimaradt Storyk/Taskok „next”-be mennek → következő sprintre

- **Sprint retrospect**

- Review után tartják, Team + Scrum Master (de általában a PO is)
- Személyes tapasztalatok/javaslatok megvitatása
- Emeljünk ki 3 dolgot → pöttyözzük az összes ember 3 dolga közül a 3 legfontosabbat
- Céges szintű problémákat SM továbbítja felfelé
- Személyes konfliktusok megbeszélése



# Scrum példa II.

- Konkrét példa: OE-NIK hallgatói fórum specifikáció
  - Lehessen kérdéseket felenni, válaszokat adni
  - Lehessen témákat indítani
  - A témák is legyenek nagyobb témákba gyűjthetők
  - Legyen pontrendszer a hasznos válaszokért
  - Legyen gamifikáció → lehessen fejleszteni az avatarunkat a pontokból
  - Lehessen hirdetni az appban
  - Lehessen tanároknak is regisztrálni
  - Lehessen a tanárok elől topicokat elrejteni
  - Lehessen képeket és videókat feltölteni
  - Lehessen fb, google, linkedin fiókkal 1 kattintással belépni

- Konkrét példa: OE-NIK hallgatói fórum specifikáció

- **Core rendszer**

- Lehessen kérdéseket felenni, válaszokat adni
    - Lehessen témákat indítani
    - A témák is legyenek nagyobb témákba gyűjthetők
    - Lehessen képeket és videókat feltölteni
    - Lehessen fb, google, linkedin fiókkal 1 kattintással belépni

EPIC

- **Gamification alrendszer**

EPIC

- Legyen pontrendszer a hasznos válaszokért
    - Legyen gamifikáció → lehessen fejleszteni az avatarunkat a pontokból

- **Ad alrendszer**

EPIC

- Lehessen hirdetni az appban

- **Teacher alrendszer**

EPIC

- Lehessen tanárokknak is regisztrálni
    - Lehessen a tanárok elől topicokat elrejteni

- **Elkészítjük a User Story-kat**

- Lehessen kérdéseket felenni, válaszokat adni
- Lehessen témákat indítani
- A témák is legyenek nagyobb témákba gyűjthetőek
- Lehessen képeket és videókat feltölteni
- Lehessen fb, google, linkedin fiókkal 1 kattintással belépní

- Ezek lesznek a Scrum Board sorai

Question  
Management

Media  
Management

Answer  
Management

Social Login

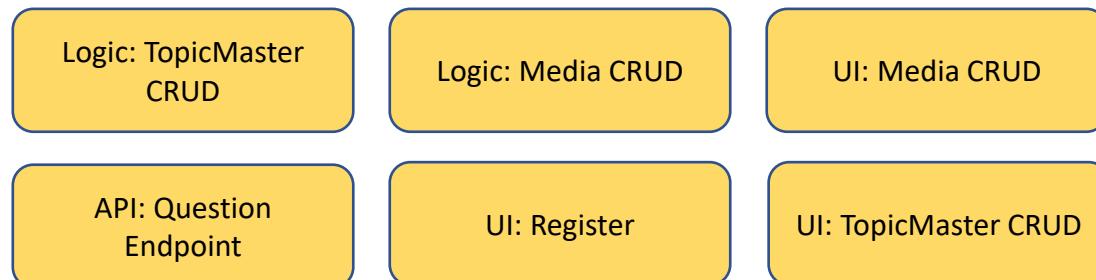
Topic Management

TopicMaster  
Management

# Core System EPIC Task készítés

26

- Question Management
- Answer Management
- Topic Management
- TopicMaster Management
- Media Management
- Social Login



DB: Question Table Implementation

DB: Images BlobStorage Implementation

Logic: Linkedin Login

API: Answer Endpoint

UI: Login

Task-ok

DB: Answer Table Implementation

DB: Vidoes BlobStorage Implementation

Logic: Question CRUD

API: Topic Endpoint

UI: Question CRUD

DB: Topic Table Implementation

Logic: FB Login

Logic: Answer CRUD

API: TopicMaster Endpoint

UI: Answer CRUD

DB: TopicMaster Table Implementation

Logic: Google Login

Logic: Topic CRUD

API: Media Endpoint

UI: Topic CRUD

# Core System EPIC Task készítés

27

- Estimation (becslés)
- Mindegyikre kerül egy pontérték
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 (Fibonacci számok)

Logic: TopicMaster CRUD 3	Logic: Media CRUD 5	UI: Media CRUD 5
API: Question Endpoint 1	UI: Register 3	UI: TopicMaster CRUD 5
DB: Question Table Implementation 1	DB: Images BlobStorage Implementation 3	Logic: Linkedin Login 0
DB: Answer Table Implementation 1	DB: Vidoes BlobStorage Implementation 3	API: Answer Endpoint 1
DB: Topic Table Implementation 1	Logic: FB Login 0	UI: Login 3
DB: TopicMaster Table Implementation 1	Logic: Google Login 0	Logic: Question CRUD 3
		API: Topic Endpoint 1
		UI: Question CRUD 5
		API: TopicMaster Endpoint 1
		UI: Answer CRUD 5
		API: Media Endpoint 1
		UI: Topic CRUD 5

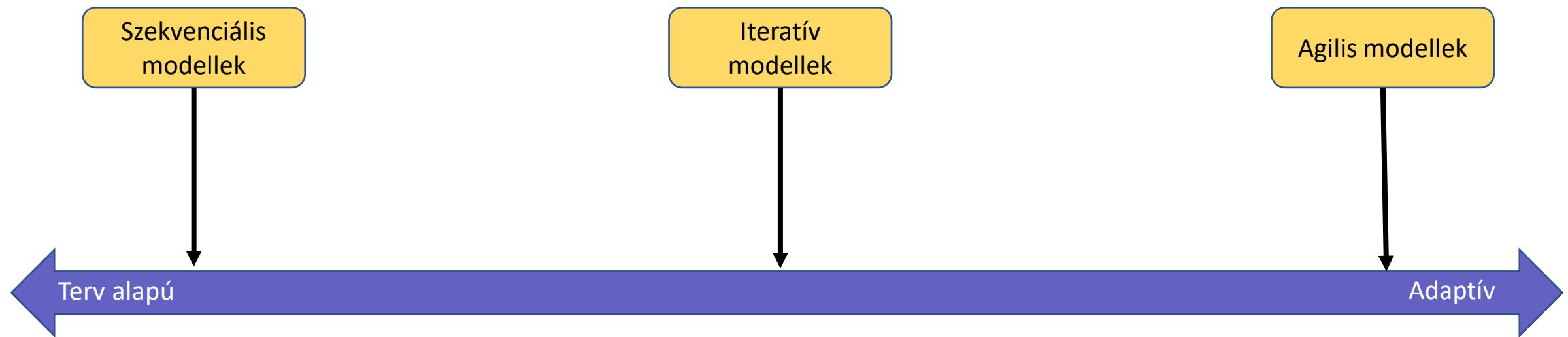
# Sprint Backlog

28

STORIES	TO DO	IN PROGRESS	IN TEST	DONE
Question Management	UI: Question CRUD	API: Question Endpoint	Logic: Question CRUD	DB: Question Table Implementation
Answer Management	UI: Answer CRUD	API: Answer Endpoint	Logic: Answer CRUD	DB: Answer Table Implementation
Topic Management	UI: TopicMaster CRUD		Logic: Topic CRUD	DB: Topic Table Implementation
TopicMaster Management		UI: Topic CRUD		API: TopicMaster Endpoint Logic: TopicMaster CRUD

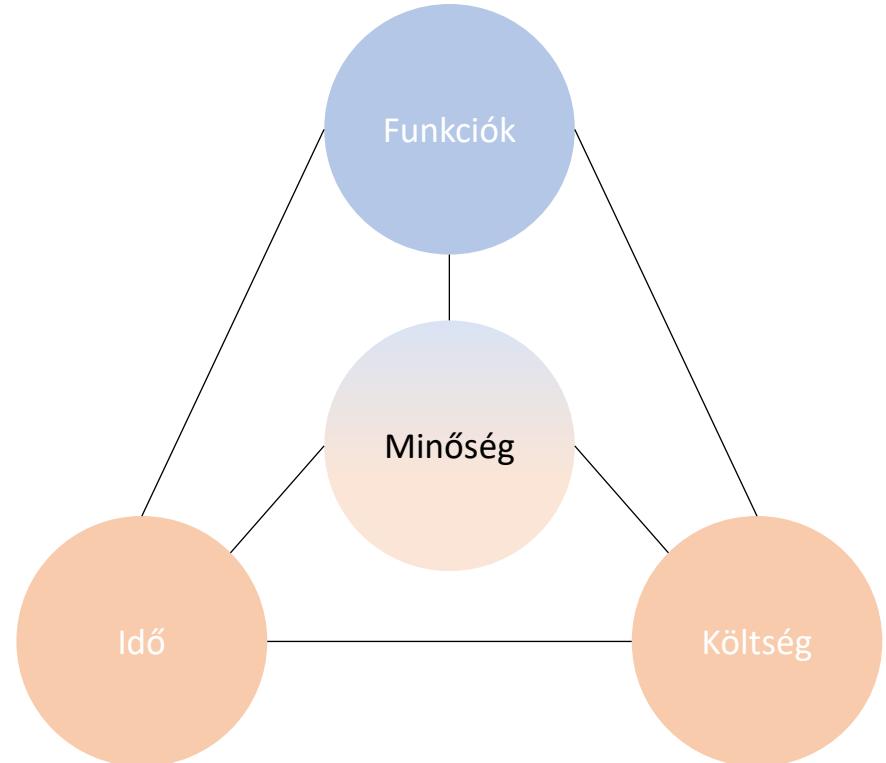
# Hagyományos vs Agilis módszerek

29

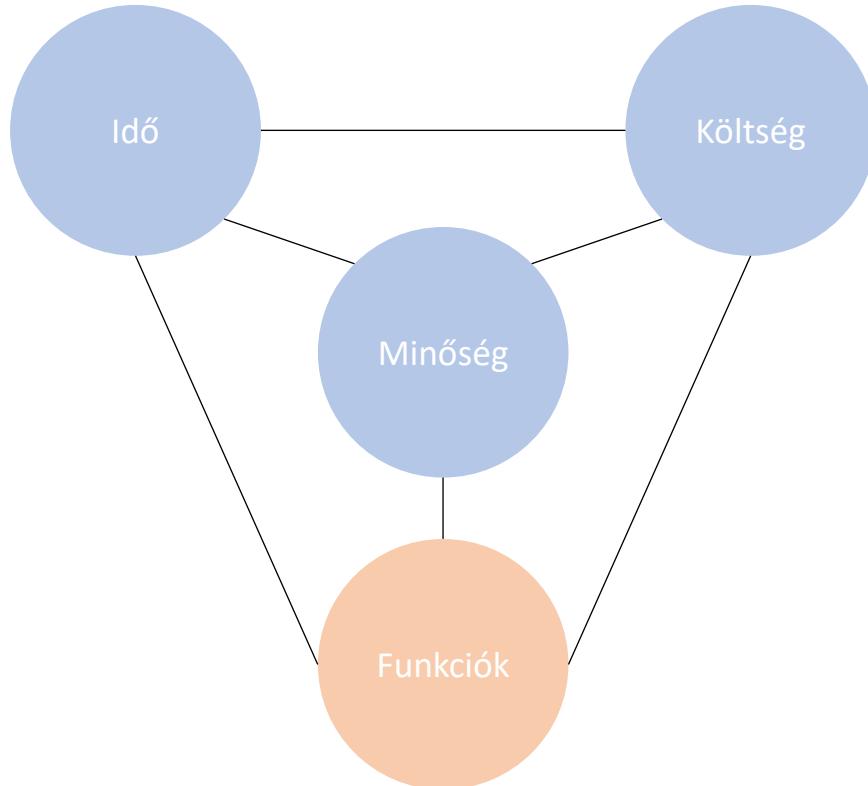


# Hagyományos vs Agilis módszerek

30



Hagyományos módszerek

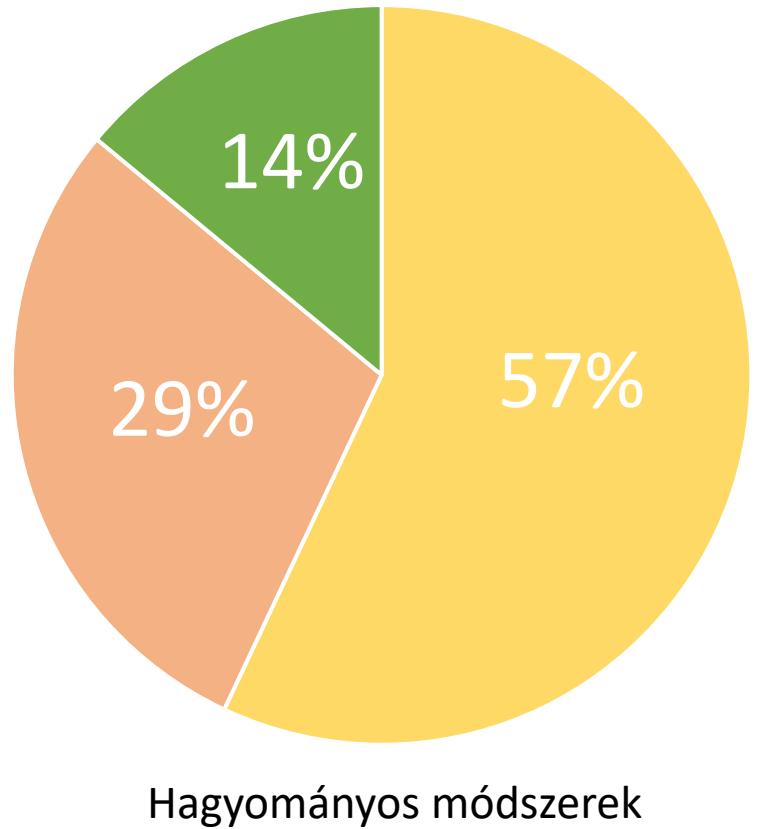


Agilis módszerek

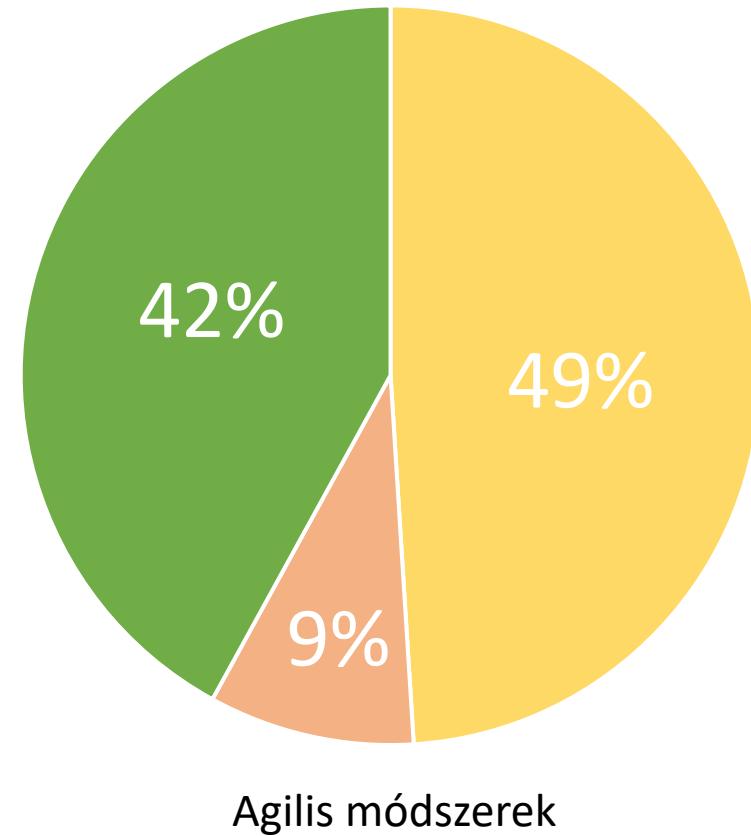
KÉK: Fix elemek  
NARANCS: Változó elemek

# Hagyományos vs Agilis módszerek

31



Sikeres projekt  
Nehézkes projekt  
Elbukott projekt



# Köszönöm a figyelmet!

Kérdés esetén e-mailben szívesen állok rendelkezésre.



# Szoftvertechnológia



ÓBUDAI EGYETEM  
NEUMANN JÁNOS INFORMATIKAI KAR

# MODUL 3

**GIT multibranch környezetben**  
**Conflict kezelés**  
**Workflows**

# Mit tudunk jelenleg?

3

- **HFT tárgyon tanultunk**

- Local és remote repository-t készíteni
- Committolni
- Pusholni a változtatásokat
- Pull-olni a github.com oldalon végzett változtatásokat
- Conflict helyzet
  - Remote és Local 1-1 fájlban különbözik → pull és push
  - Remote és Local ugyanabban a fájlban különbözik → conflict kezelés szükséges

- **Mit szeretnénk elérni?**

- Csoportmunkában szeretnénk dolgozni
- Hatékonyan szeretnénk a kódokat megosztani egymással

- **Cél**

- Eltérünk a fejlesztés menetétől, anélkül, hogy elrontanánk
- Párhuzamos „valóságokat” készítünk, majd néha egyesítünk

- **Branch**

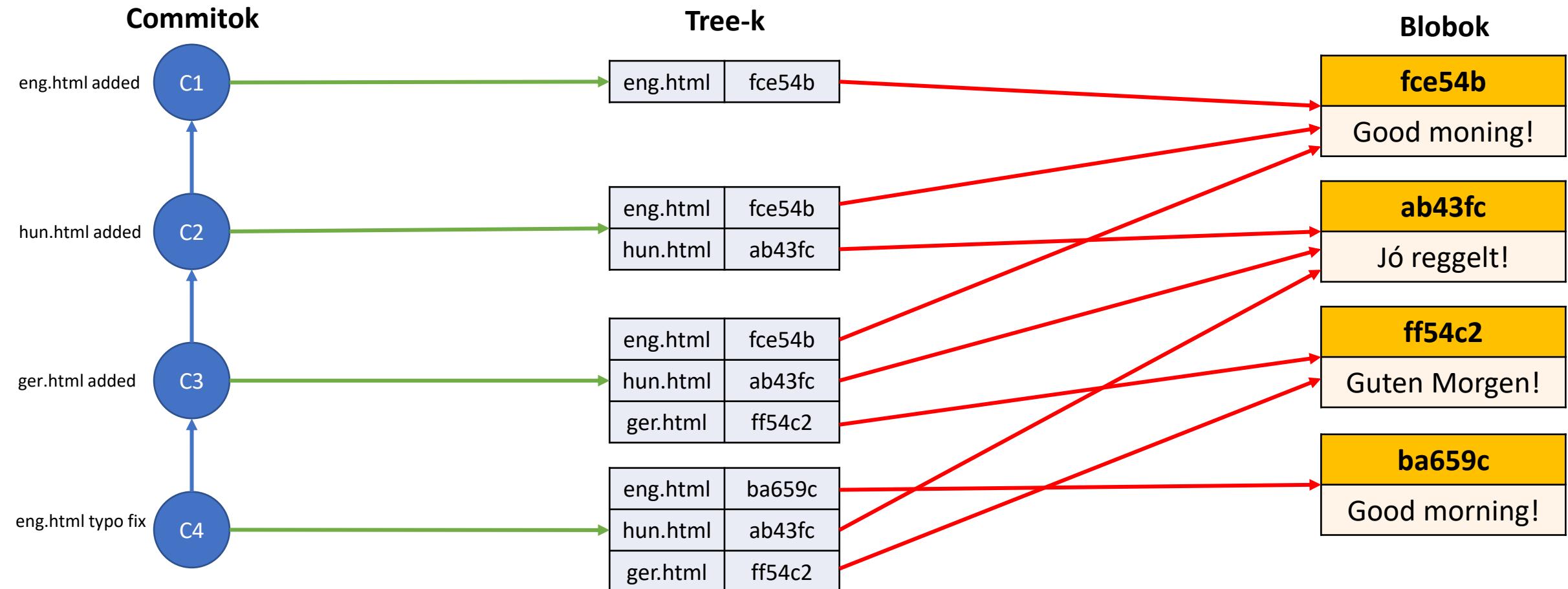
- Egy egyszerű pointer, ami egy commitra mutat
- Alapból létezik egy master
- minden committolásnál eggyel arrébb megy

- **HEAD**

- Szintén egy pointer, ami az aktuális local commitra mutat
- A working directory ebben az állapotban van



# Repository belső felépítése

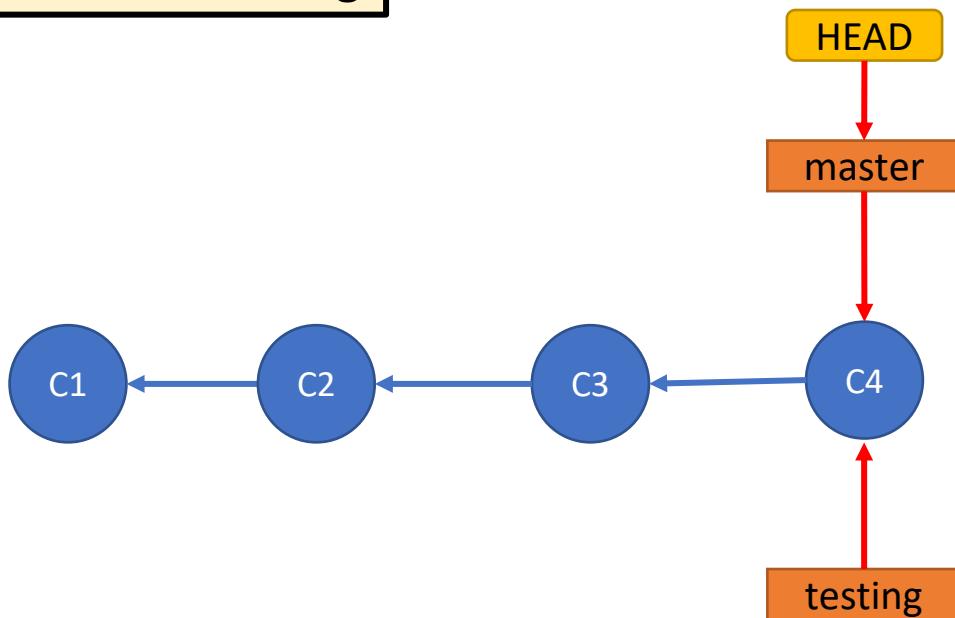


Commit tartalma: referencia az ōsre/ōsökre és a tree-re

# Branch létrehozás

- Legyen egy testing nevű branch is a master mellett

```
git branch testing
```



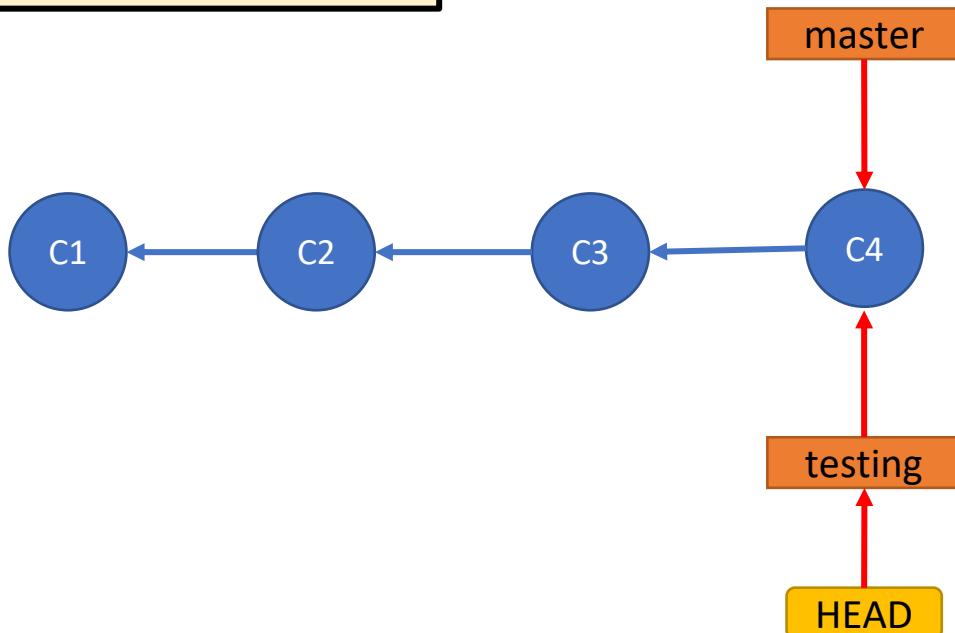
- Az aktuális állapotba helyezi a testing mutatót

# Branch váltás

7

- Álljunk át a testing branchre

```
git checkout testing
```



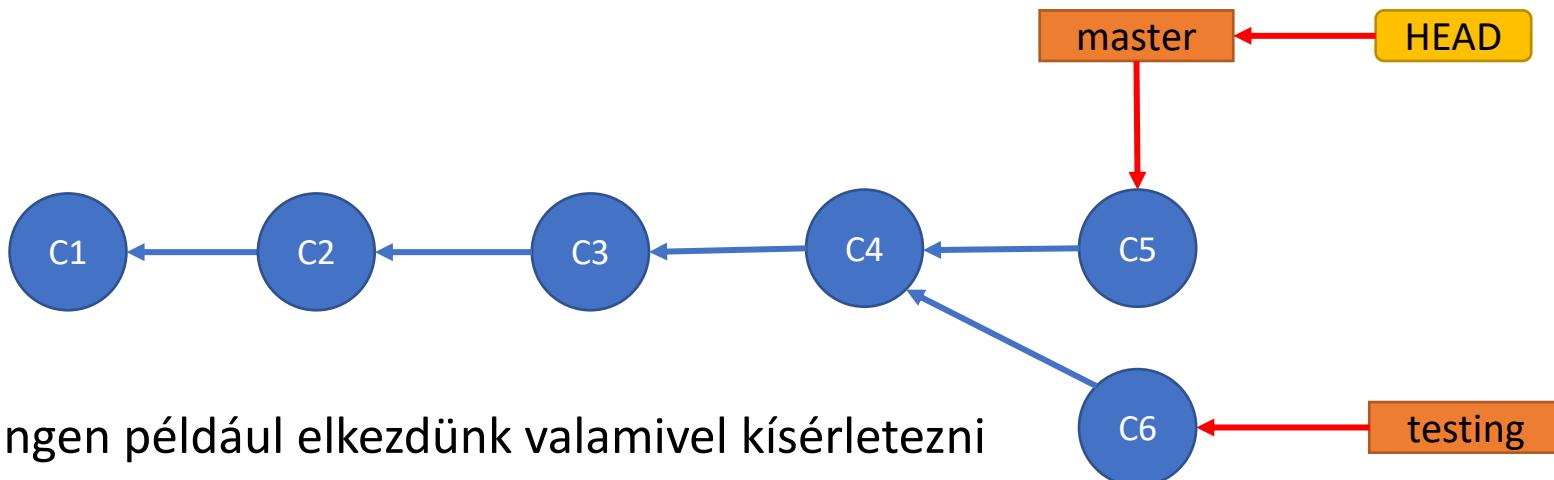
- Branch létrehozás és checkout egyben

```
git checkout -b testing
```

- A workdir nem változik meg, mert jelenleg minden branch ugyanoda mutat
- De ezen a ponton elindulhatunk két irányba a két branch-el

# Branch váltás

- Mindkét branchen csinálunk 1-1 commit-ot



- A testingen például elkezdünk valamivel kísérletezni
- A masteren viszont haladunk tovább pl. formázásokkal
- A logban szeretnénk szépen látni ezt az egészet**

```
git log --oneline --decorate --graph --all
```

- all helyett bármelyik branch neve is beírható

## 1. Szükség van egy új feature-re

1. Leágazunk a fő branchről egy új feature branchhel
2. Itt dolgozunk a feature-ön

## 2. Kapunk egy hotfix kérést

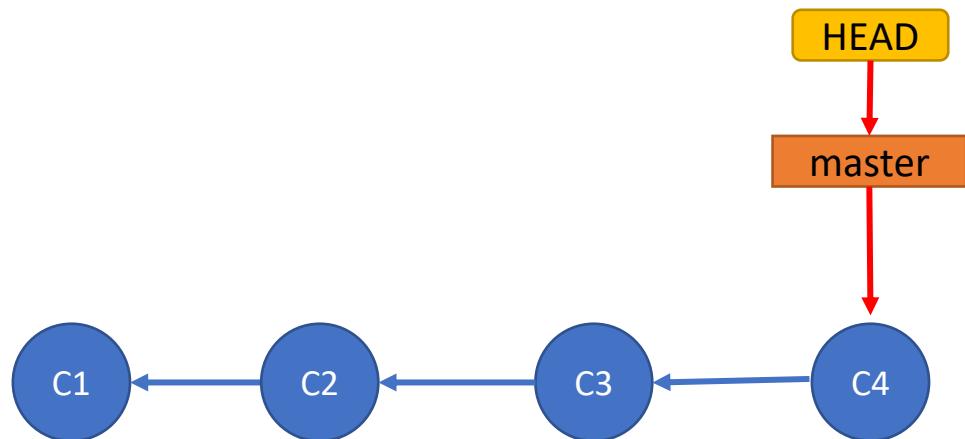
1. Leágazunk a fő branchről egy új hotfix branchhel
2. Itt megoldjuk
3. Tesztelés után belefésüljük (merge) a fő branchbe

## • Fő branch??

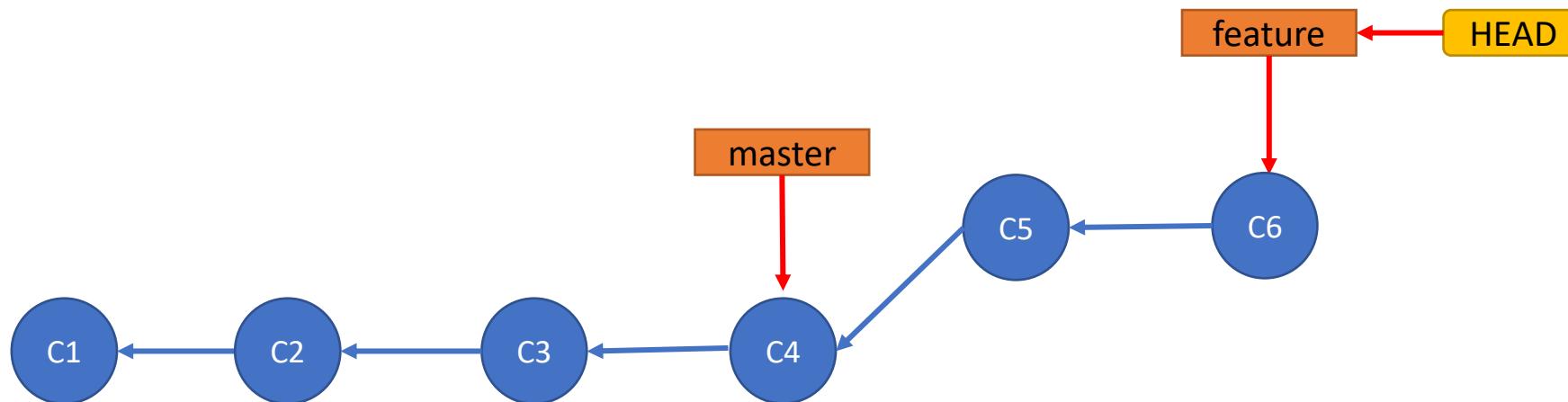
- Ez alapvetően lehetne a master (példában így lesz)
- De általában egy development branchról ágazunk le
- Masterre a release verziókat visszük vissza



- Jelenleg itt tartunk most



- Új feature miatt leágazunk és committolunk oda párat

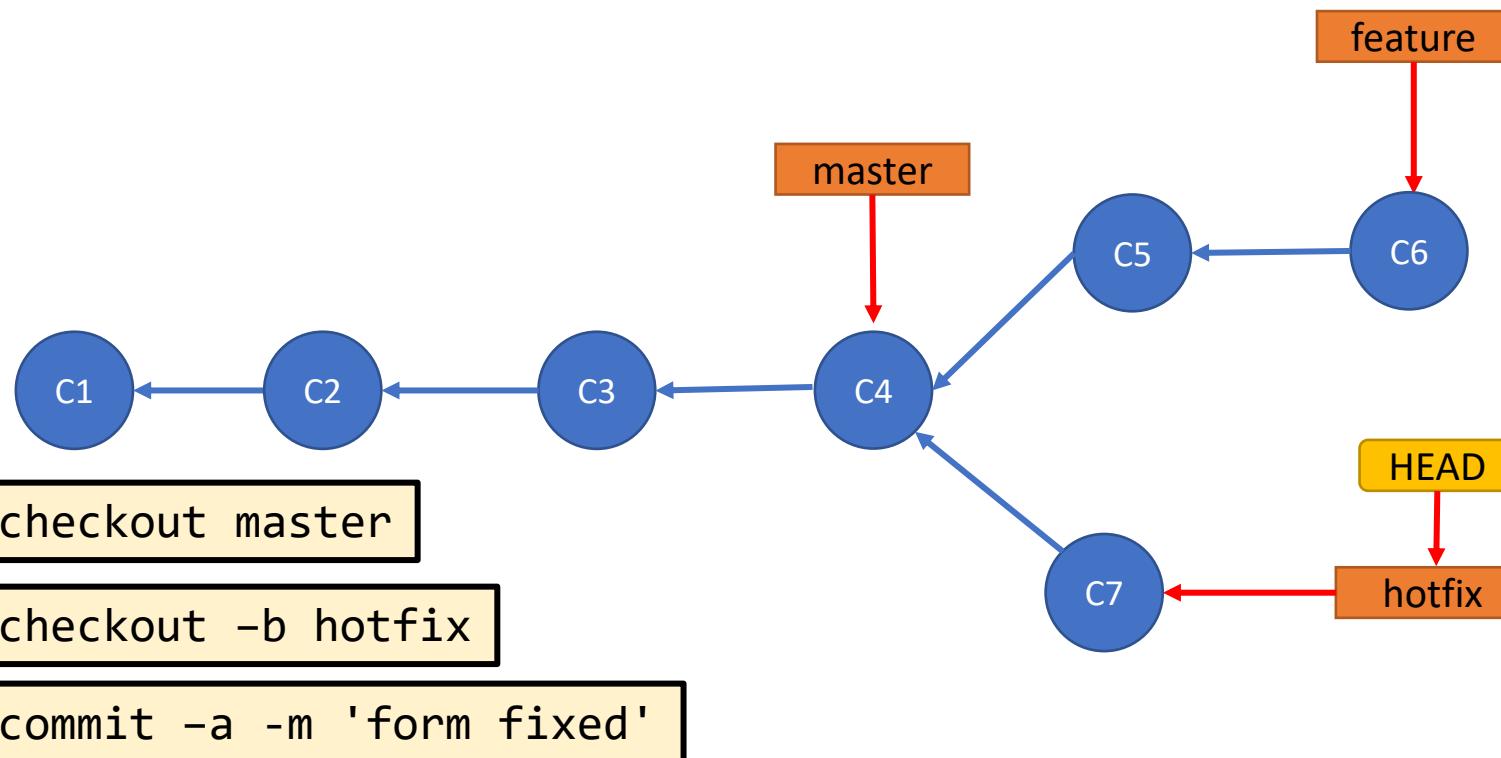


```
git checkout -b feature
```

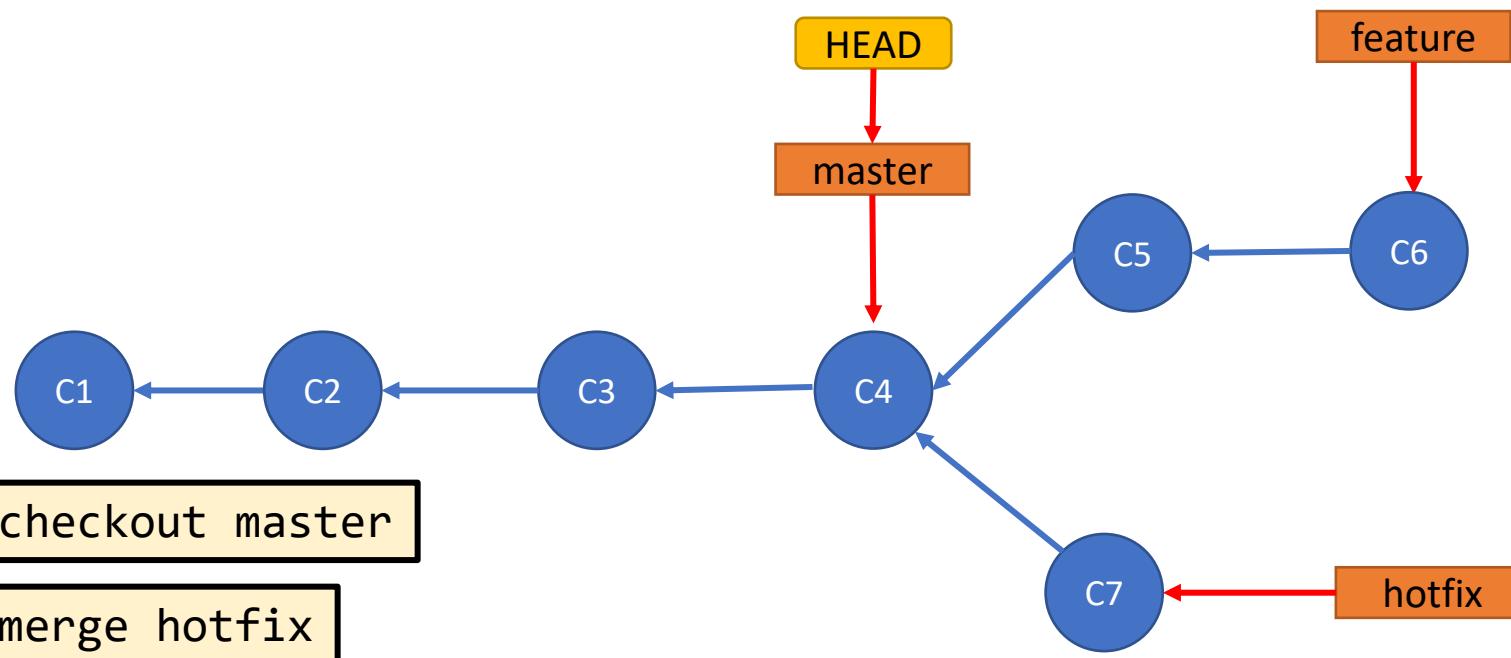
```
git commit -a -m 'feature design'
```

```
git commit -a -m 'feature done'
```

- Megérkezik a hotfix kérés az eddigi „éles” munkára

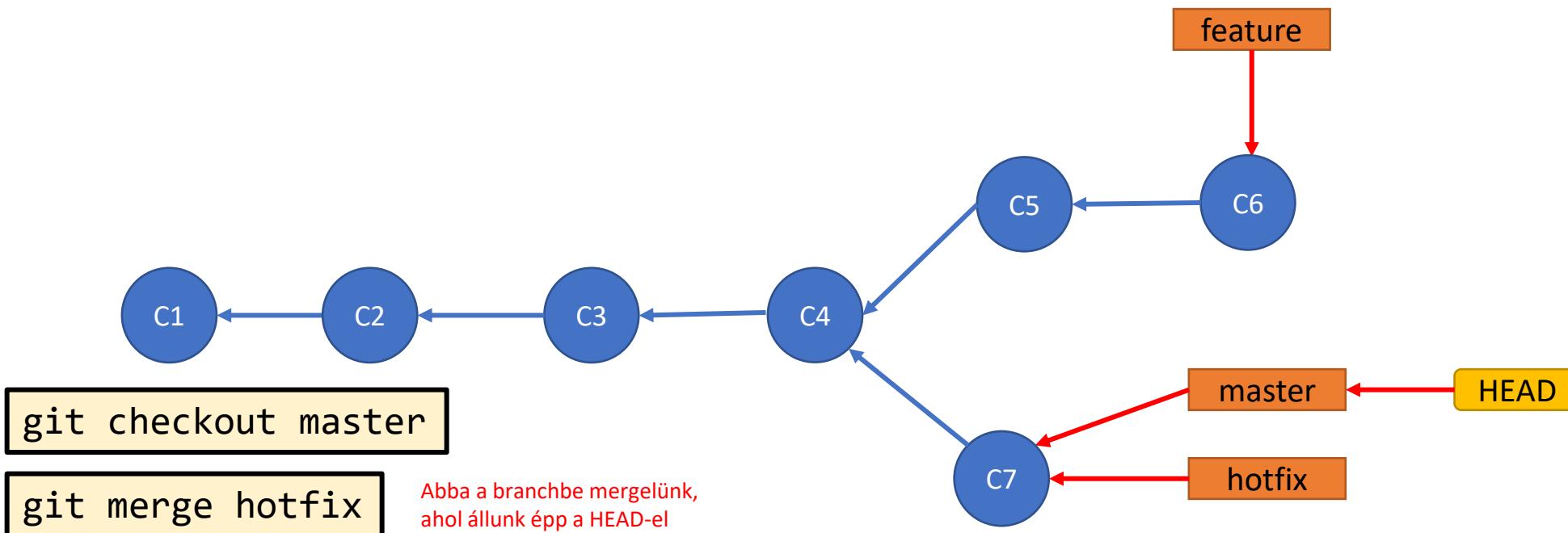


- Be kellene mergelni a masterbe a hotfixet
  - Mivel a master azóta nem haladt, ez nagyon egyszerű



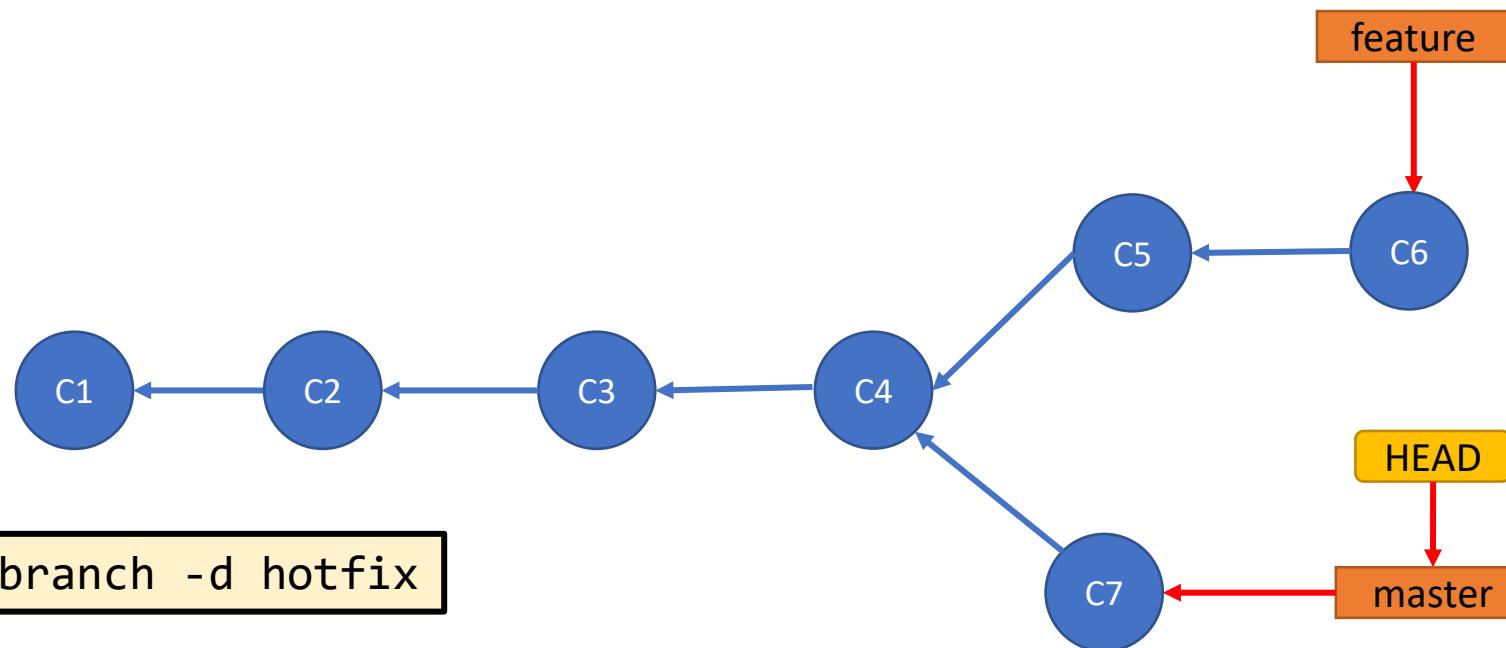
Mivel a master nem haladt, ezért a hotfix végére eltolható a master pointer (fast-forward).  
Persze így a feature mergelése lesz majd nehéz, de ez mindenki következik...

- Be kellene mergelni a masterbe a hotfixet
  - Mivel a master azóta nem haladt, ez nagyon egyszerű

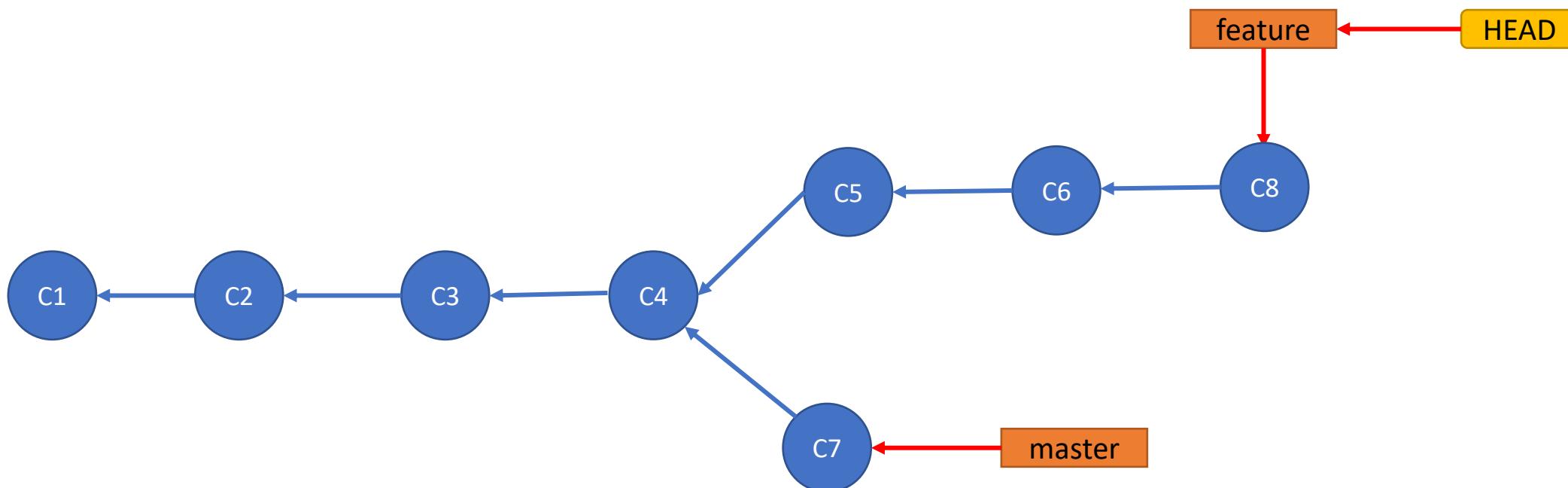


Mivel a master nem haladt, ezért a hotfix végére eltolható a master pointer (fast-forward).  
Persze így a feature mergelése lesz majd nehéz, de ez mindenki következik...

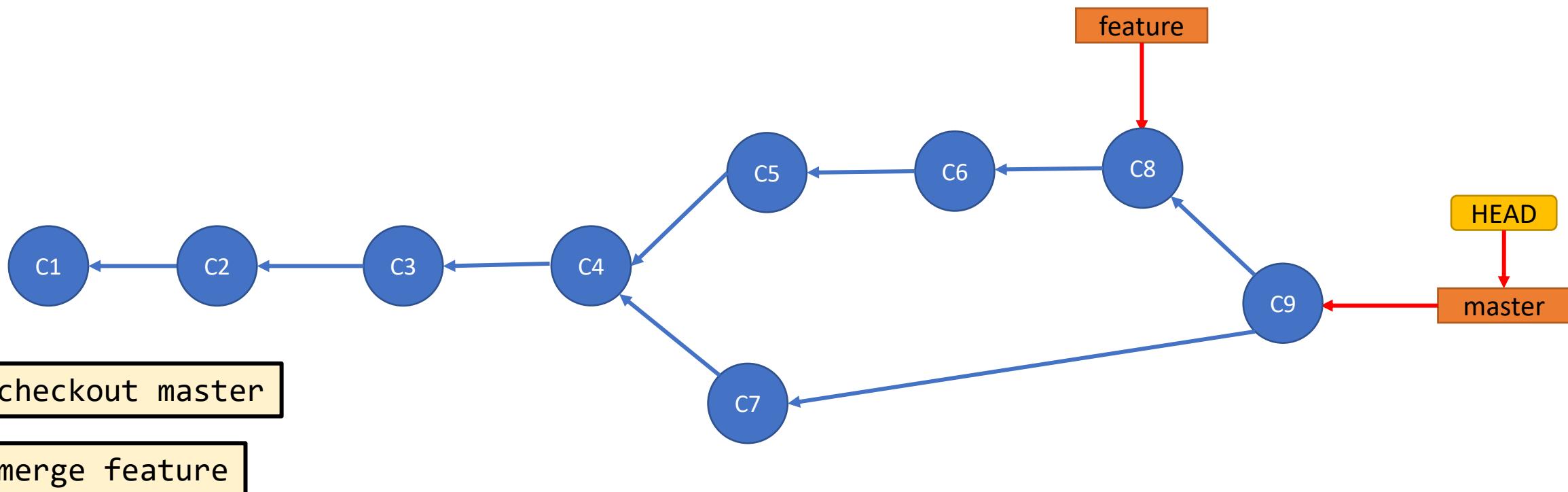
- A hotfix branch akár törölhető is már
  - Nem szerencsés, ha elveszik ez a „történet”



- A feature fejlesztése folytatódik, 1 commit után elkészül
  - A feature-t be kell mergelni a master-ba

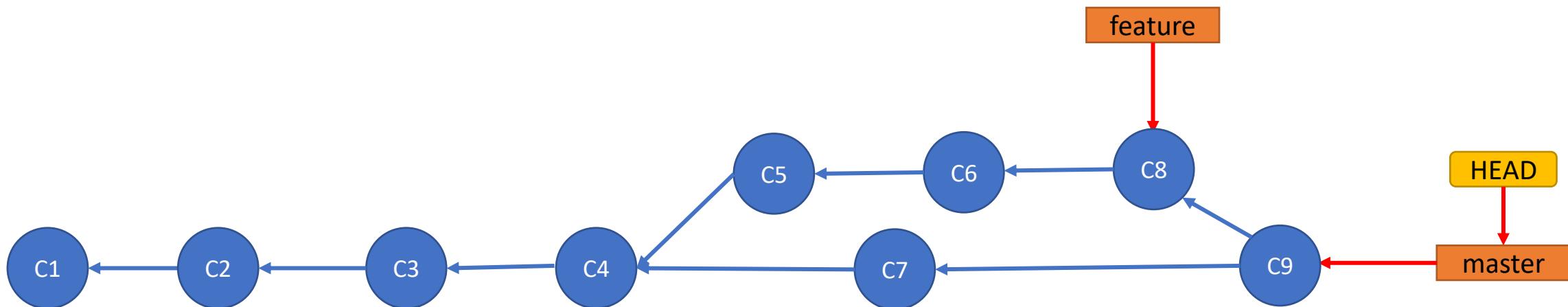


- A feature fejlesztése folytatódik, 1 commit után elkészül
  - A feature-t be kell mergelni a master-ba



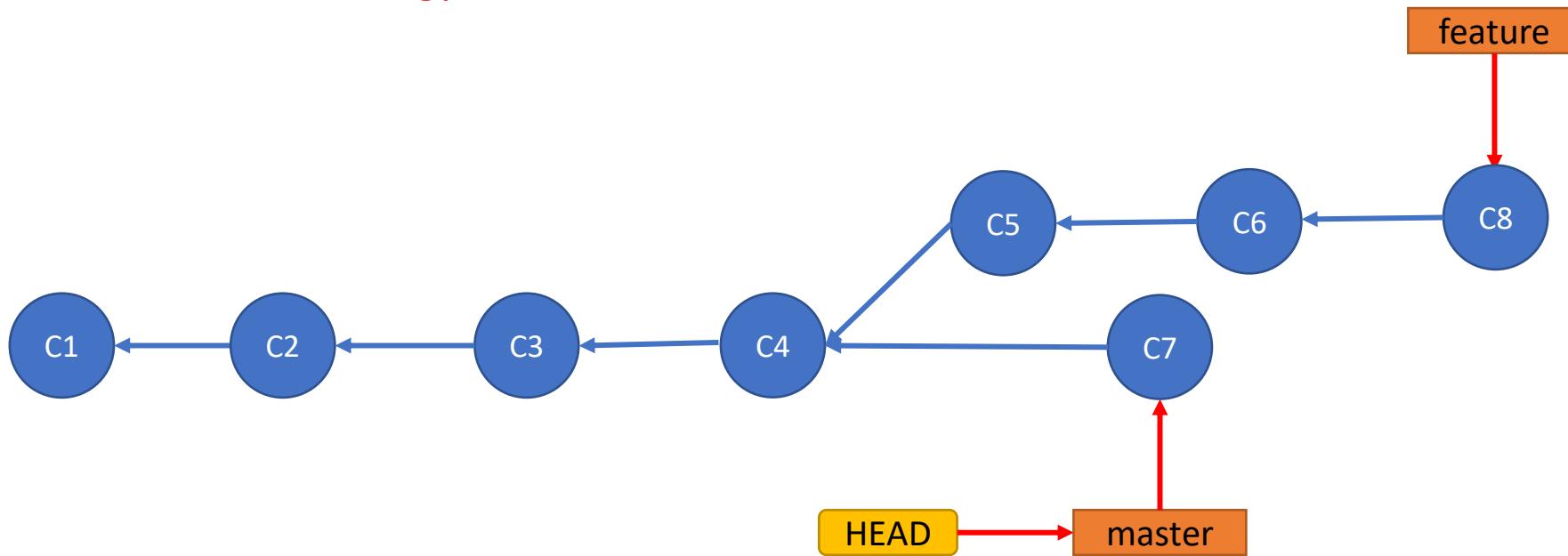
Itt most C4 a közös ős-commit. A merge csak úgy lehetséges, hogyha egy commitban egyesül a két ág (C9 = merge commit két őssel)

- A feature branch akár törölhető
  - C9 commitban minden megvan, amit fejlesztettünk benne



*Ha netán a hotfix kellett volna a feature további fejlesztéséhez, akkor a mastert be kellett volna mergelni a feature-be, majd a feature-t tovább fejleszteni és végül őt bemergelni.*

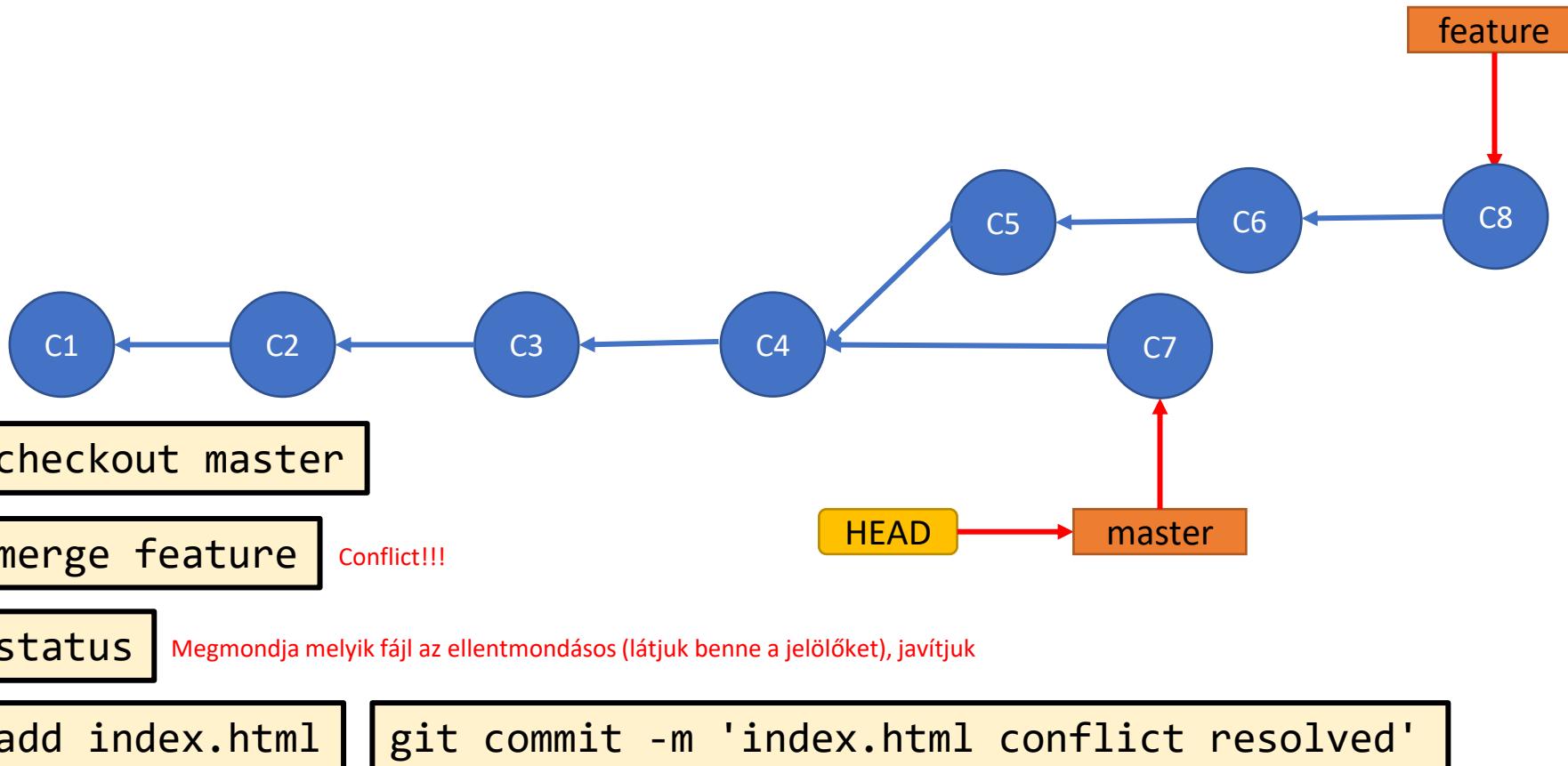
- A hotfix és a feature nem mondott egymásnak ellent
    - De mi lenne, hogyha még a merge commit előtt lennék
    - Valamint C8 és C7 egymásnak ellent mond?



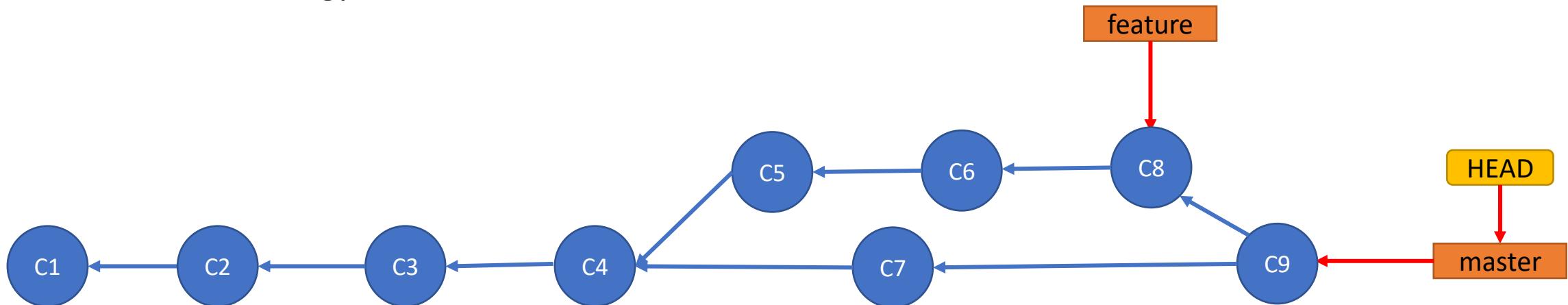
**Ellentmondás:** ugyanazt a fájlt módosítja

(elviekben ha az egyik a fájl 23. sorát, a másik a fájl 65. sorát módosítja, az nem baj)

- A hotfix és a feature nem mondott egymásnak ellent
  - De mi lenne, hogyha még a merge commit előtt lennének
  - Valamint C6 és C7 egymásnak ellent mond?



- A hotfix és a feature nem mondott egymásnak ellent
  - De mi lenne, hogyha még a merge commit előtt lennének
  - Valamint C6 és C7 egymásnak ellent mond?



git checkout master

git merge feature

Conflict!!!

git status

Megmondja melyik fájl az ellentmondásos (látjuk benne a jelölőket), javítjuk

git add index.html

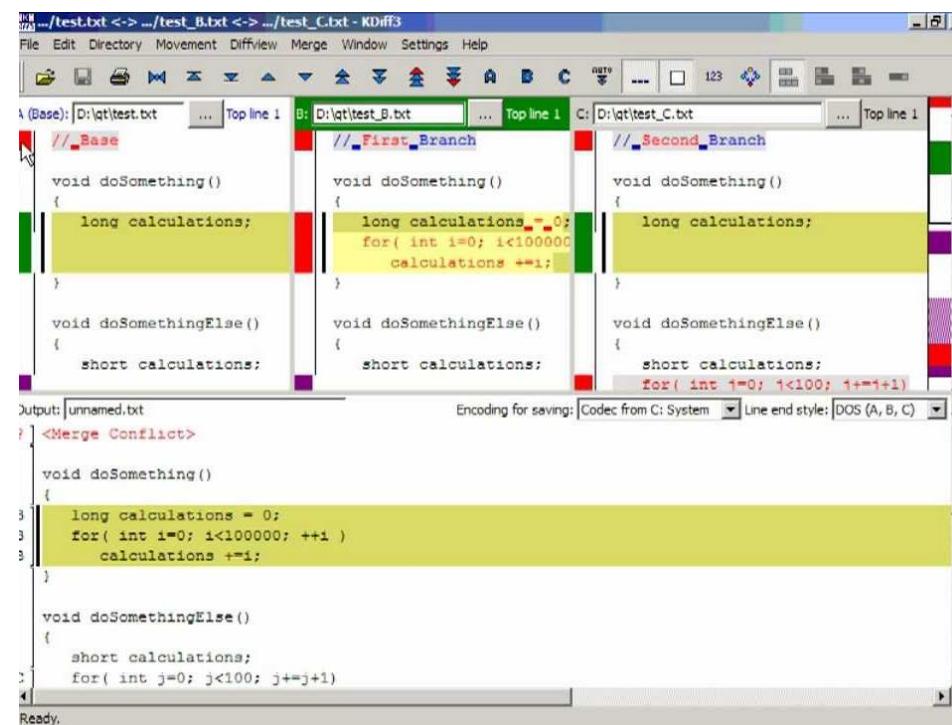
git commit -m 'index.html conflict resolved'

# Merge tool

22

- A conflictok feloldását nem csak kézzel lehetjük meg
  - Javasolt GUI: Kdiff3 (*letöltés + path-ba helyezés*)
  - A **merge** és **add** parancsok között futtatjuk

git mergetool



git config --global merge.tool kdiff3

git config --global mergetool.kdiff3.path "C:/kdiff3.exe"

git config --global mergetool.kdiff3.trustExitCode false

- Összes branch listázása

```
git branch
```

- Összes branch listázása + utolsó commit message-ek

```
git branch -v
```

- Szűrő feltételek

```
git branch --merged
```

```
git branch --no-merged
```

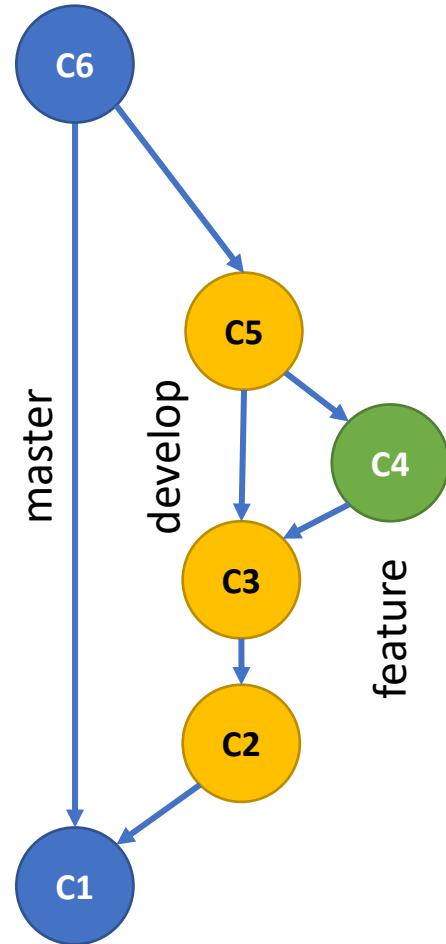
- no-merged: nagyon hasznos („befejezetlen ügyek”)



# Branching workflows

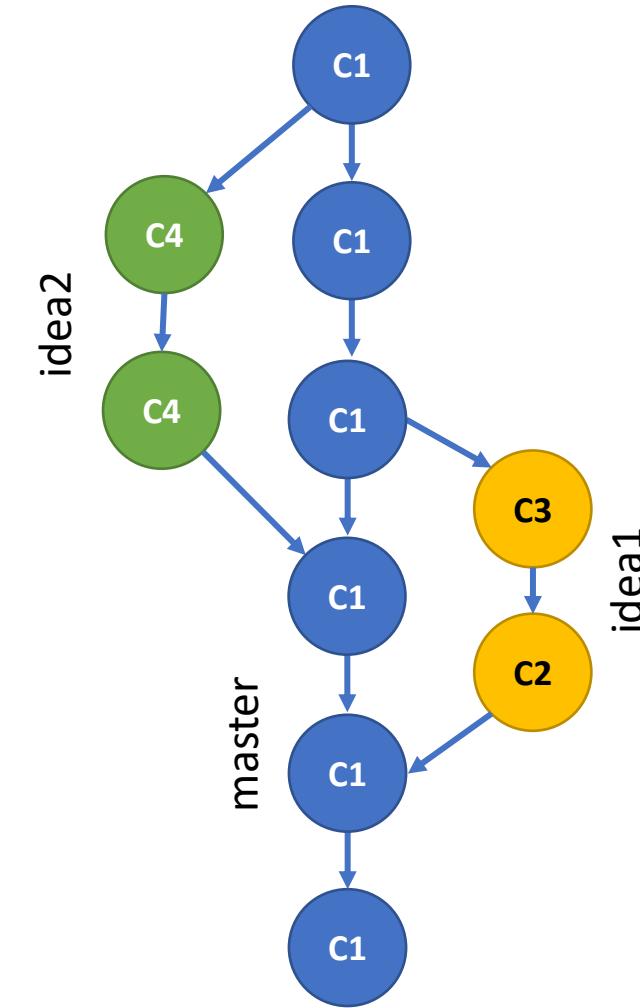
24

## Long-running branches/GitFlow



- **Long-running branches**
  - A master mindig stabil
  - A developon dolgozunk
  - Feature fejlesztéshez leágazunk
  - Developra mergelünk vissza
    - Nem használunk fast-forwardot!
    - Mindig merge commit!
  - Masteren lévő merge commitok release pontok
- **Topic branches**
  - Masteren fejlesztünk
  - Ötleteket próbálunk ki a brancheken
  - Feature-öket fejlesztünk a brancheken
  - Néha eldobunk 1-1 branchet, mert rossz ötlet volt, amit kipróbáltunk ott

## Topic branches/Trunk-based



# Köszönöm a figyelmet!

Kérdés esetén e-mailben szívesen állok rendelkezésre.



# Szoftvertechnológia



ÓBUDAI EGYETEM  
NEUMANN JÁNOS INFORMATIKAI KAR

# **MODUL 4**

**UML diagramok célja**

**Viselkedési diagramok**

**Strukturális diagramok**

**Nem UML diagramok**

# Mi a baj az eddigi modellekkel?

3

- Továbbra is kommunikációs szakadék van a megrendelő és a fejlesztők között
  - Agilis bevonással enyhíthető, hamarabb kibuknak hibák
  - Prototípusok fejlesztése – véleményezése is hasznos
  - De még mindig nincs meg a közös nyelv
- Továbbra is kommunikációs szakadék van fejlesztő és fejlesztő között
  - Tapasztalattól, tudásszinttől függően is előfordul, hogy nem egy nyelvet beszélnek
  - Nehéz szóban értekezni arról, hogy a Logic milyen interfészen át lesz elérhető
  - Írásban kódegyeztetés a tervezési fázisban?
    - C# kód? Gyakran ezt csináljuk, de akkor már implementálunk is egyben
    - Valaki nagy lendülettel nekikezd, a többiek pedig próbálnak hozzáírogatni
    - Legyen valamilyen absztrakt, egyszerűsített nyelvünk a tervezéshez!
      - Pszeudokód! → Nem alkalmas bonyolultabb nyelvi elemek ábrázolására (interfész, esemény, delegált, reflexió, stb...)
      - UML diagrammok!

- UML: Unified Modelling Language
  - Grafikus leírónyelv, amely segíti az alábbi folyamatokat
    - Vizualizálás
    - Specifikálás
    - Tervezés
    - Dokumentálás
- Kinek jó ez?
  - Megrendelő egy folyamatábrát könnyen tud értelmezni
  - Fejlesztő könnyebben megérти, hogy a másik fejlesztő rendszere hogy működik
  - A vizuális ábrázolás jobb megértést biztosít
  - Dokumentáció és kellően alapos lesz általa
- (Egyetemi fejlesztési szakdolgozatba is kötelező)

- UML elvileg egy szigorú modellező nyelv
- De ma már elhagyható minden, ami nem szükséges (agilis UML)
- Modellező eszközök:
  - Microsoft Visio
  - Visual Paradigm for UML
  - Rational Rose
  - Enterprise Architect (legelterjedtebb, de fizetős)
  - DIA

# UML diagram típusok

- **Viselkedési diagramok (behavioral diagram)**

- Használati eset diagram (use-case diagram)
- Aktivitás diagram (activity diagram)
- Állapotgép diagram (state machine diagram)
- Interakciós diagram (interaction diagram)
  - Kommunikációs diagram (communication diagram)
  - Időzítés diagram (timing diagram)
  - Szekvencia diagram (sequence diagram)

- **Kinézet terv (wireframe) ← nem UML**

Előzetes tervezés fázis

- **Strukturális diagramok (structure diagram)**

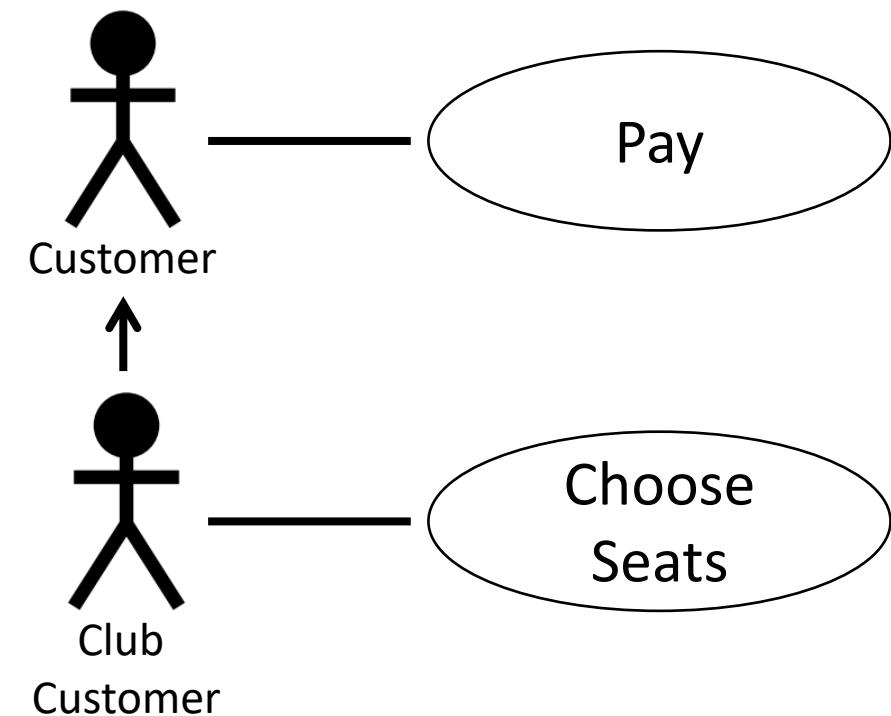
- Osztály diagram (class diagram)
- Komponens diagram (component diagram)
- Objektum diagram (object diagram)
- Kompozit-struktúra diagram (composite structure diagram)
- Telepítési diagram (deployment diagram)
- Csomag diagram (package diagram)

Részletes tervezés /  
Megvalósítás fázis

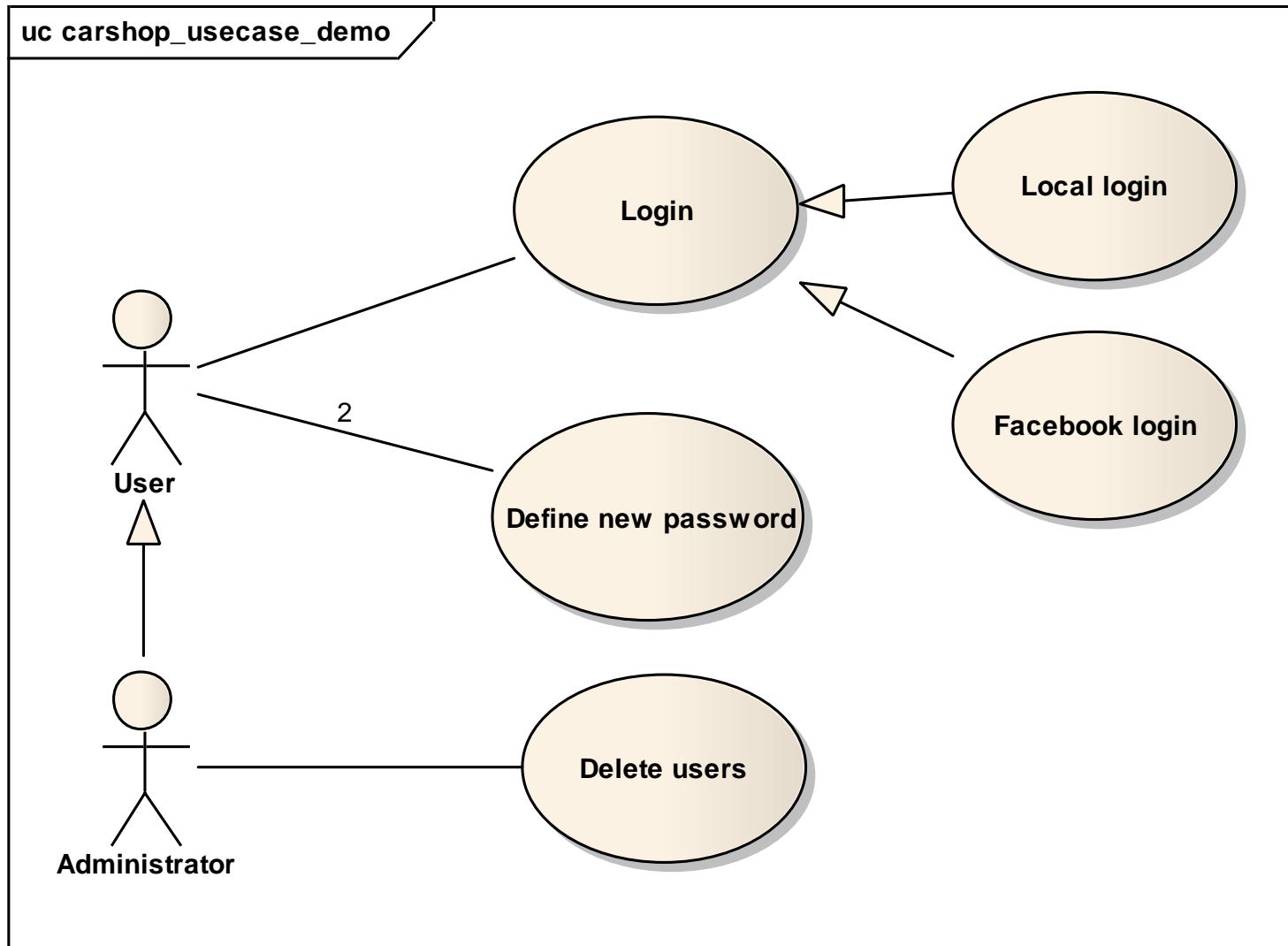
# Use-case diagram

7

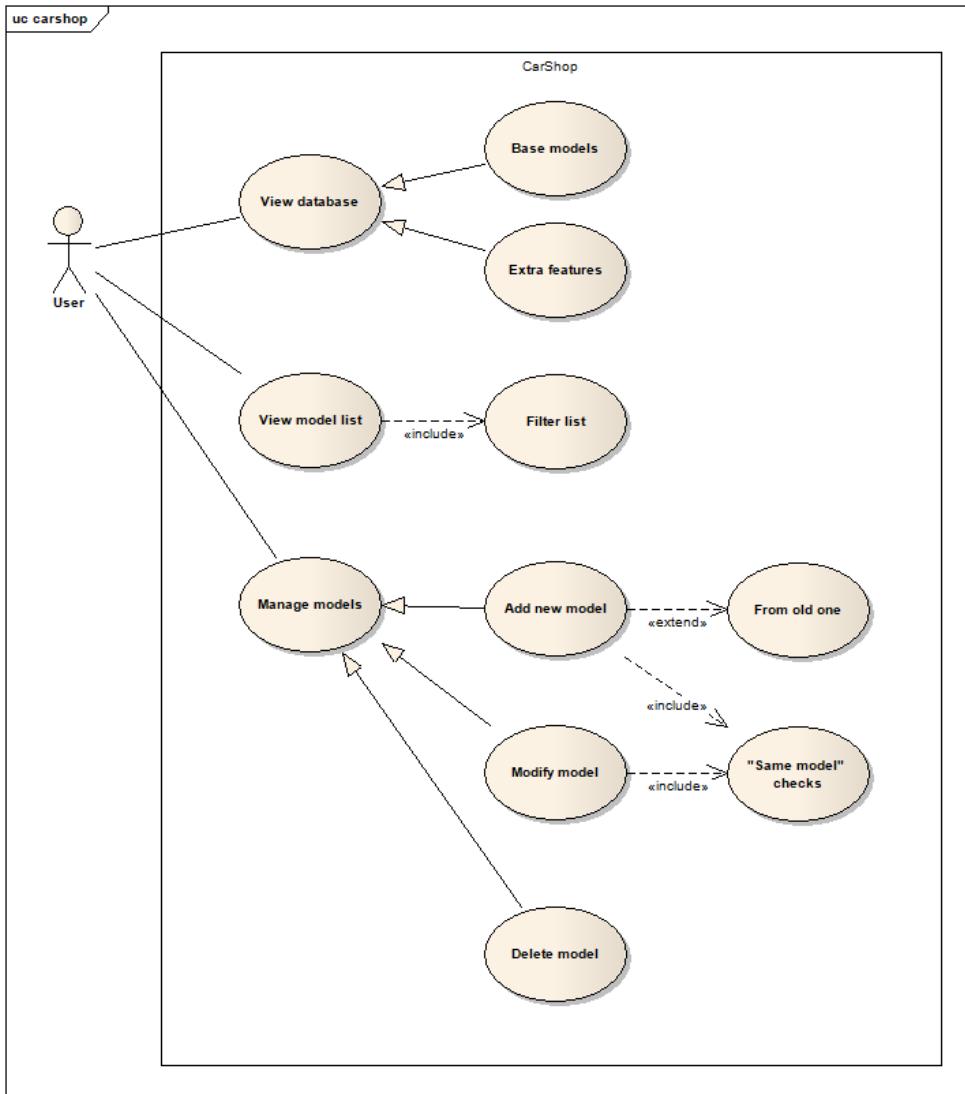
- Milyen jogosultsággal, milyen funkciók vehetőek igénybe?
  - Aktor és használati esett közötti megfeleltetés
  - Öröklődés engedélyezett
    - Aktorok között
    - Használati esetek között
  - Számosság (aktor és használati eset között)
  - Tartalmazás/kibővítés
    - Használati esetek között



# Use-case diagram



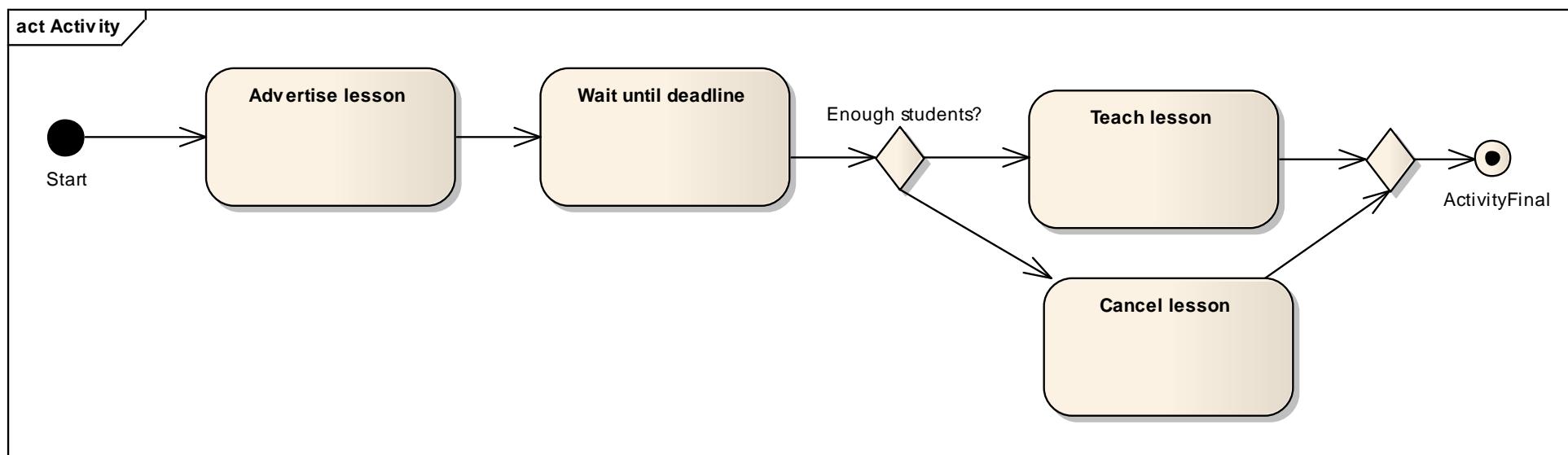
# Use-case diagram



# Activity diagram

10

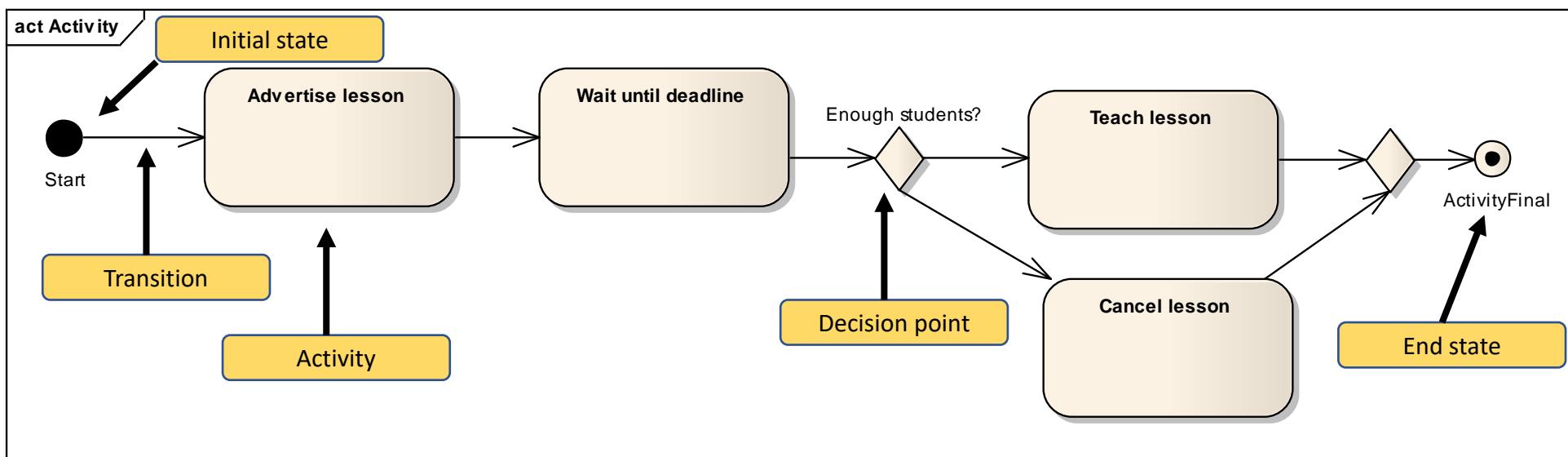
- Use-case utáni következő lépés
- Használati eseteknél a cél
  - Belső folyamatok ábrázolása
  - Egymás után következők ábrázolása
- 1 használati eset → 1 Activity diagram
- Egy diagramon belül más use-case-ek is előjöhetnek



# Activity diagram

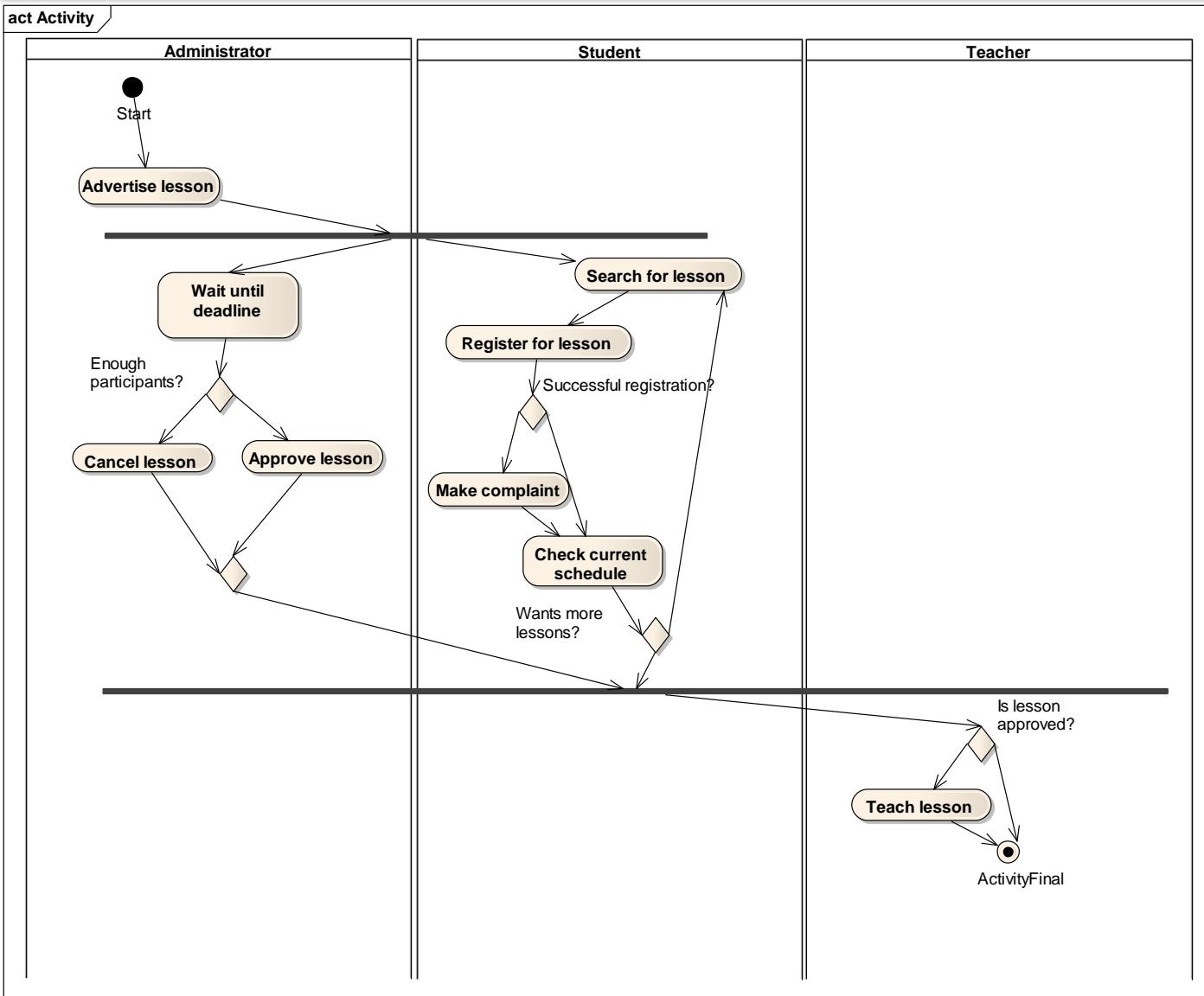
11

- Use-case utáni következő lépés
- Használati eseteknél a cél
  - Belső folyamatok ábrázolása
  - Egymás után következőségek ábrázolása
- 1 használati eset → 1 Activity diagram
- Egy diagramon belül más use-case-ek is előjöhetnek



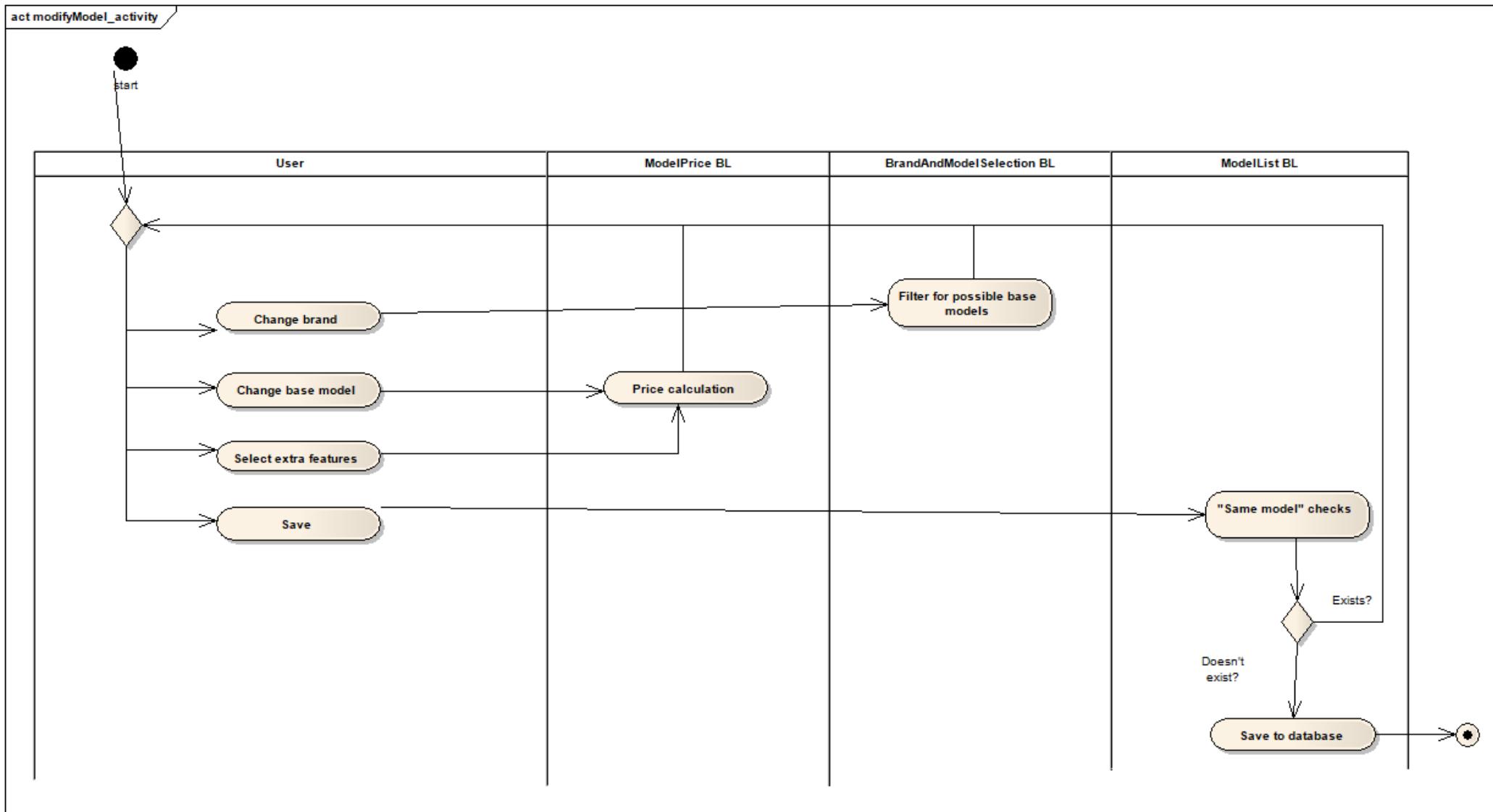
# Activity diagram

12



# Activity diagram

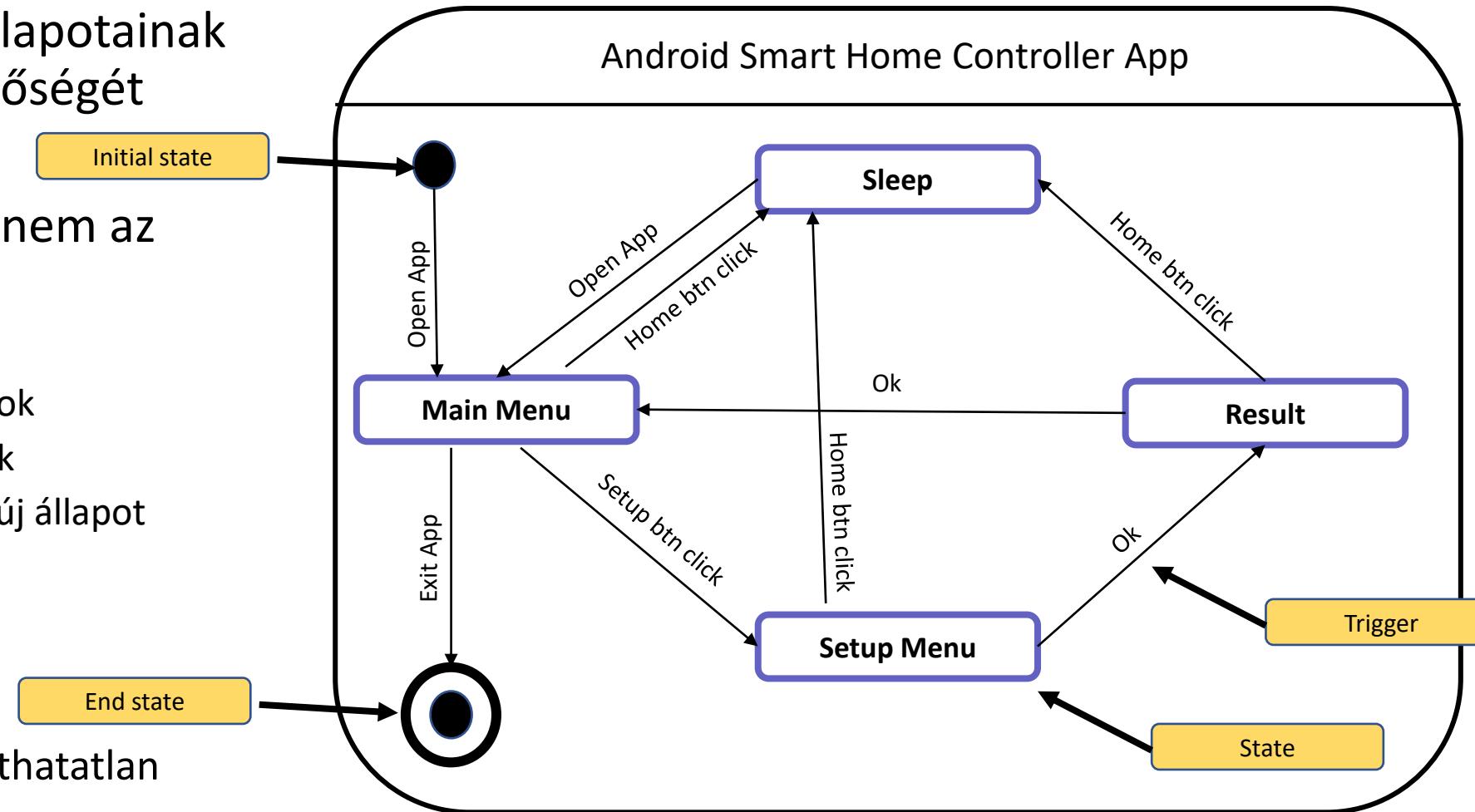
13



# State diagram

14

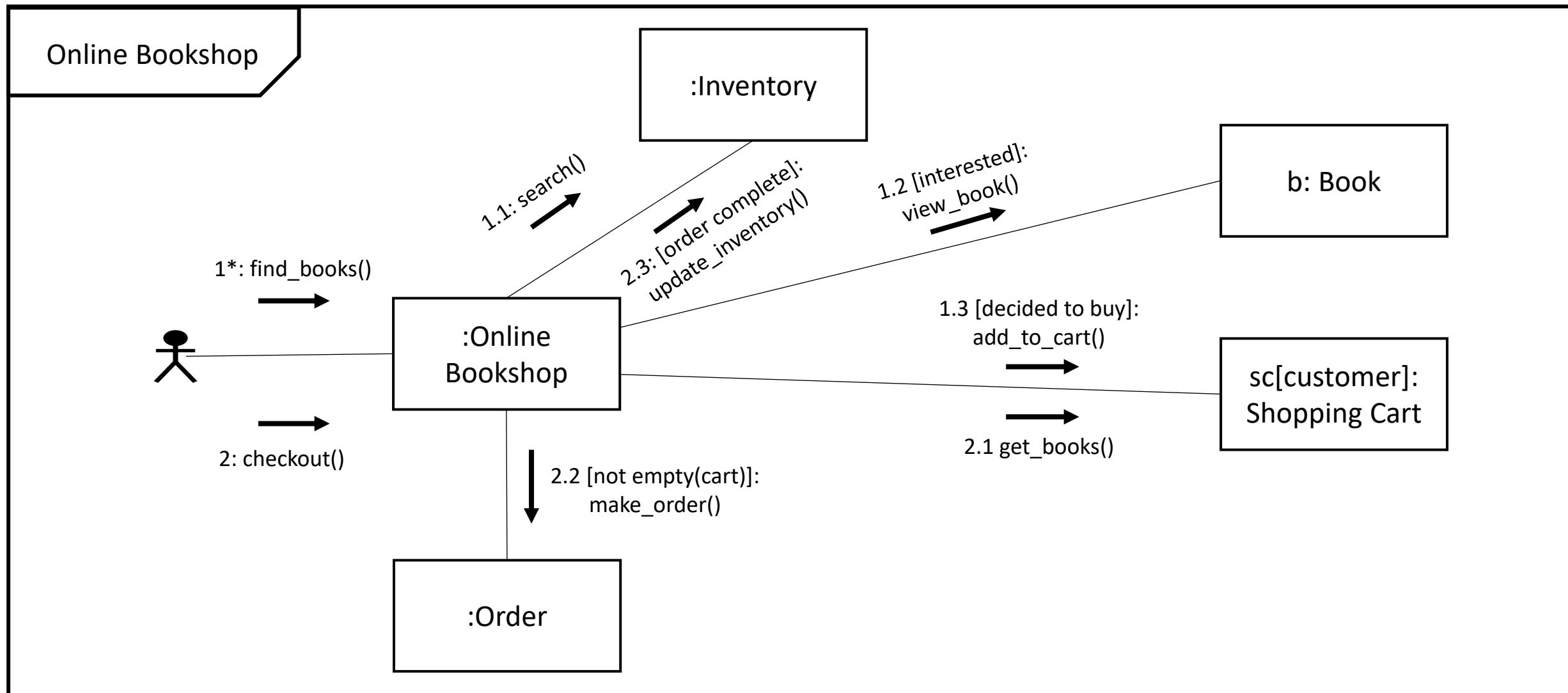
- Rendszer/objektum állapotainak egymás után következőségét ábrázolja
- Véges állapotgépekre nem az egyetlen eszköz
  - State transition table
    - Első oszlop: állapotok
    - Első sor: események
    - Cellák: feltételek + új állapot
  - Petri hálók
  - SDL state machine
- Probléma:
  - Már ez a méret is átláthatatlan



- Viselkedési diagramok utolsó kategóriája
- Egy folyamat résztvevői (aktorok vagy modulok vagy rétegek) közötti kommunikációs folyamatokat ábrázoljuk
- Hárrom különböző diagramot különböztet meg
  - **Communication diagram:** résztvevők és sorrend
  - **Timing diagram:** időzítési információk, megkötések és állapotok is
  - **Sequence diagram:** ciklusok, feltételek és élettartamok is
- Mindhárrom típus ugyanarra a célra való csak más mélységben mutatja be a kommunikációs folyamatokat

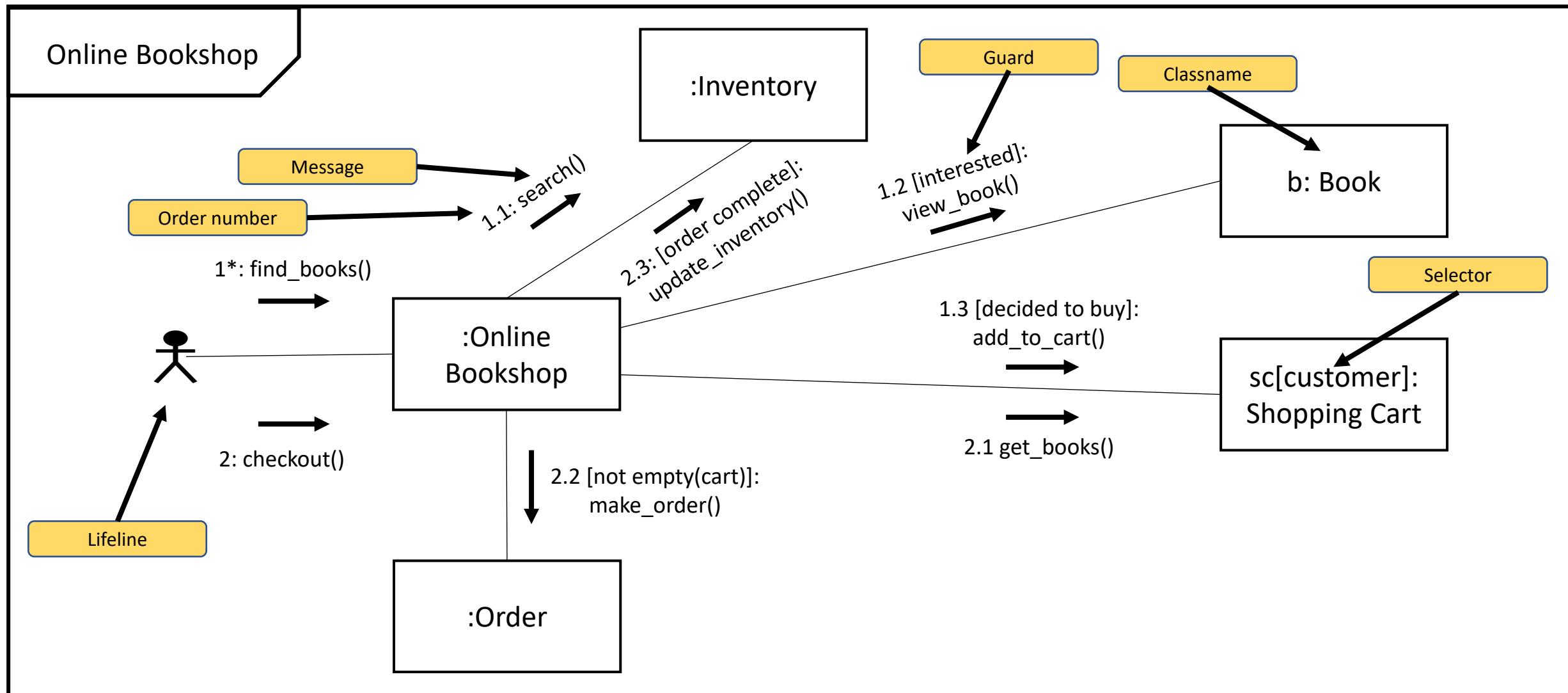
# Communication diagram

16



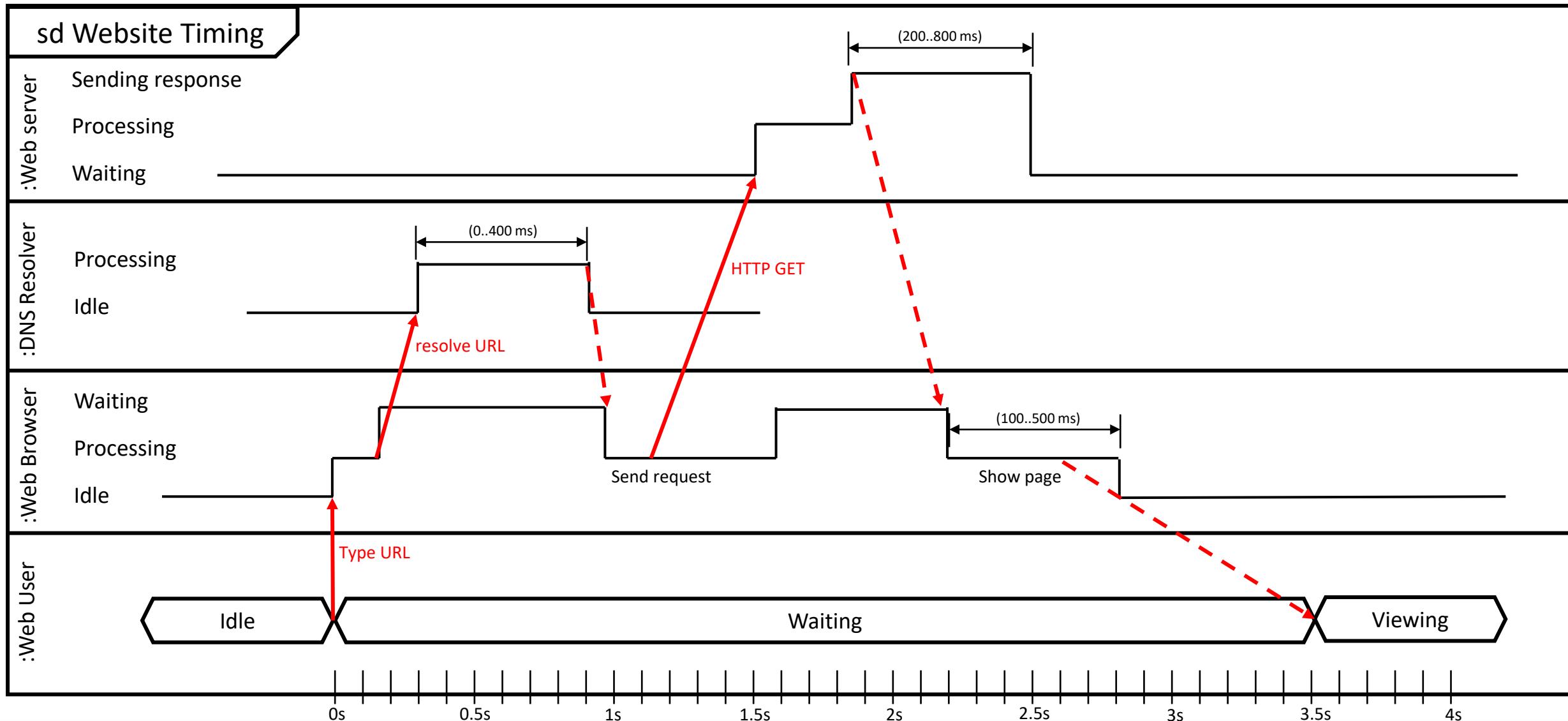
# Communication diagram

17



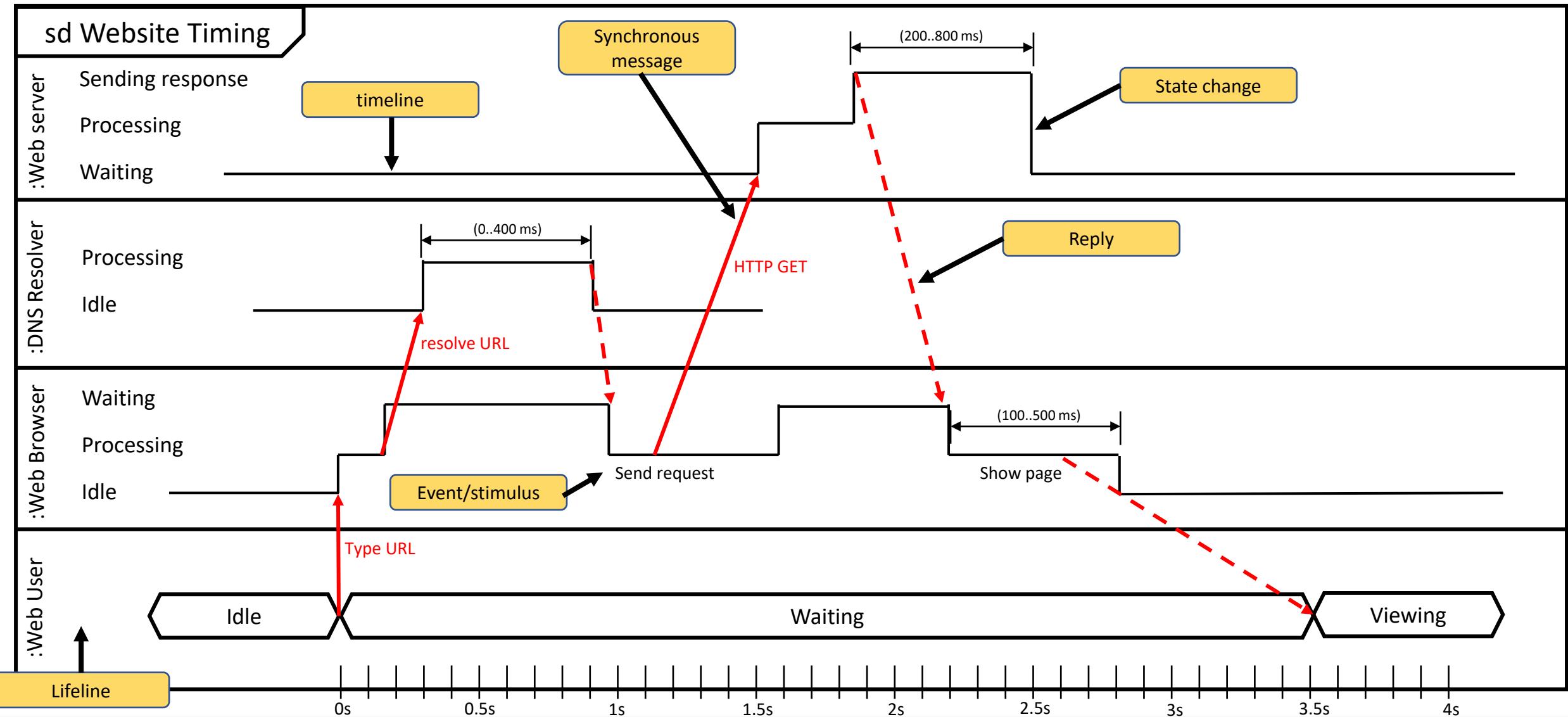
# Timing diagram

18



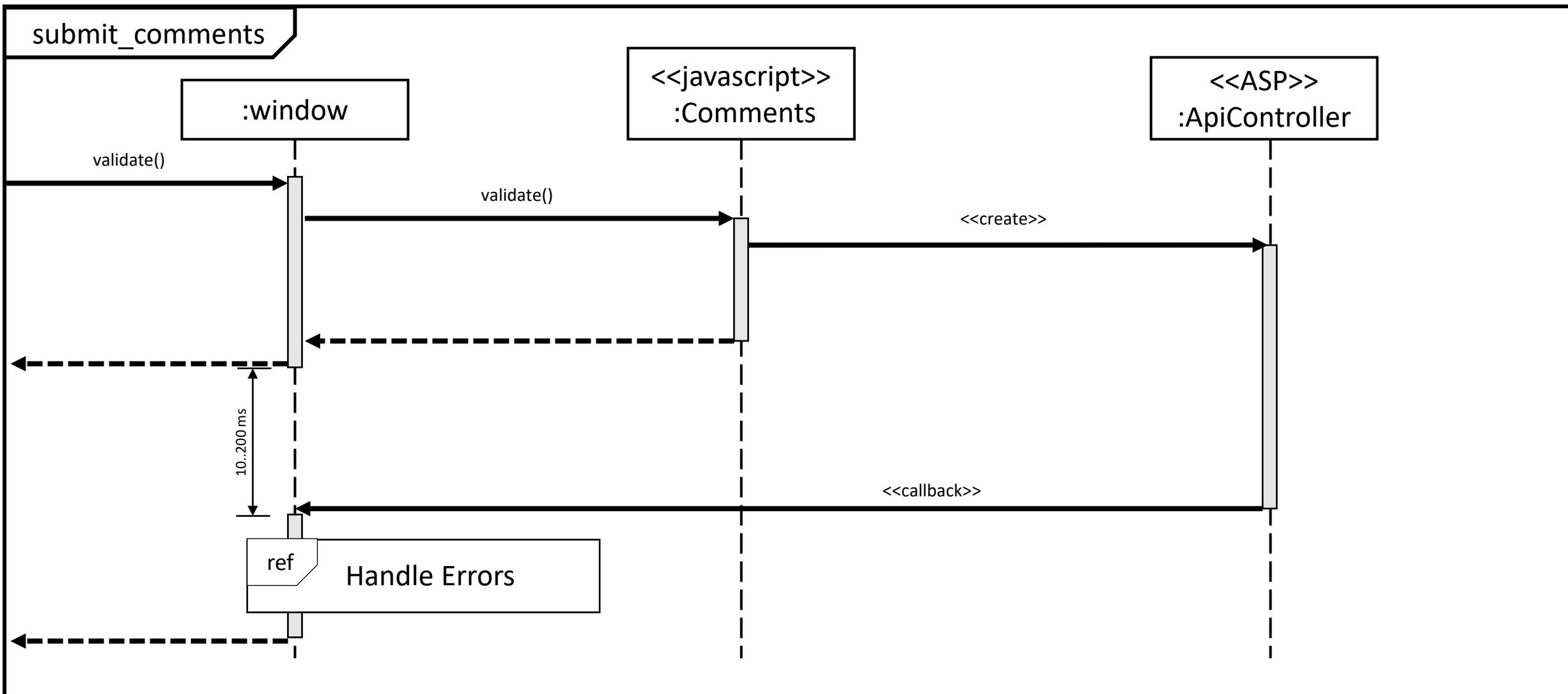
# Timing diagram

19



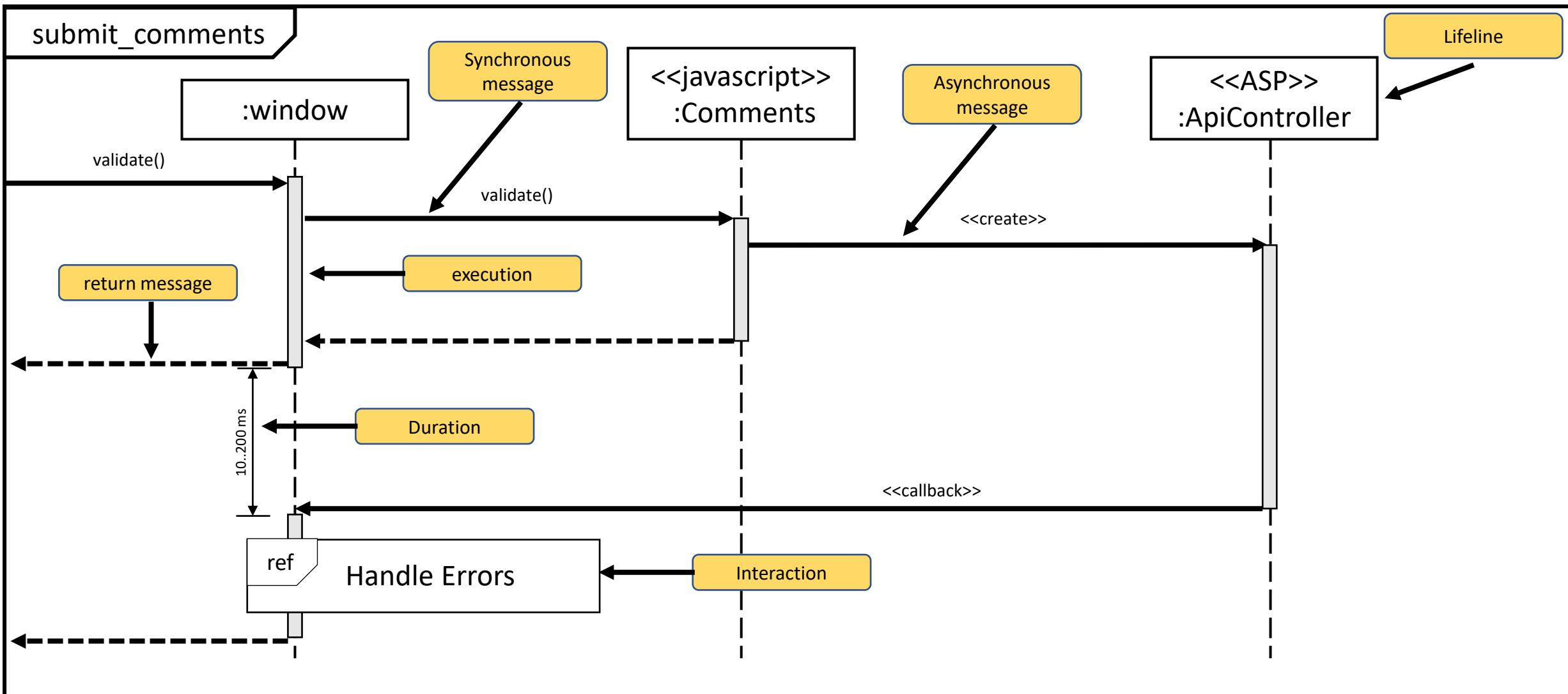
# Sequence diagram

20

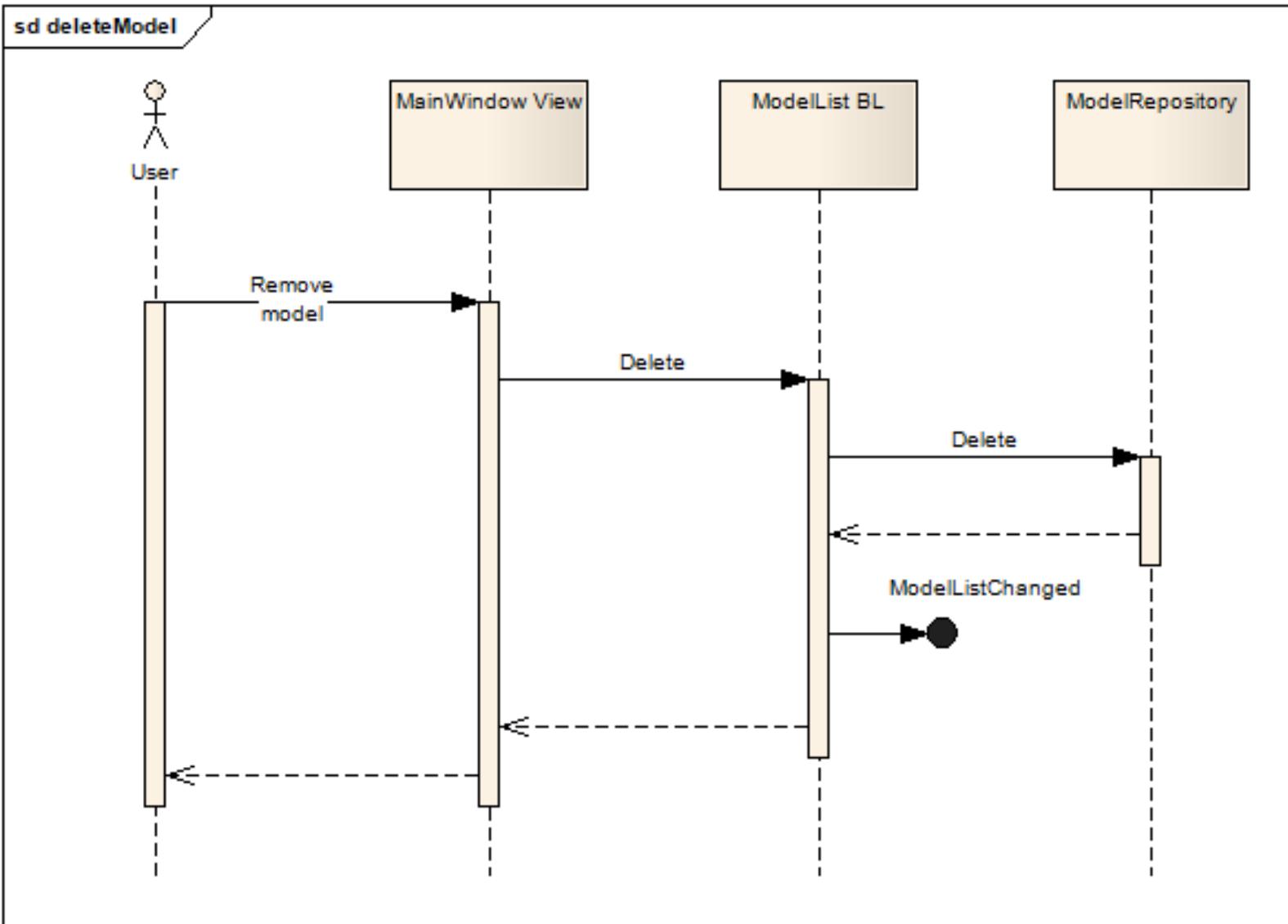


# Sequence diagram

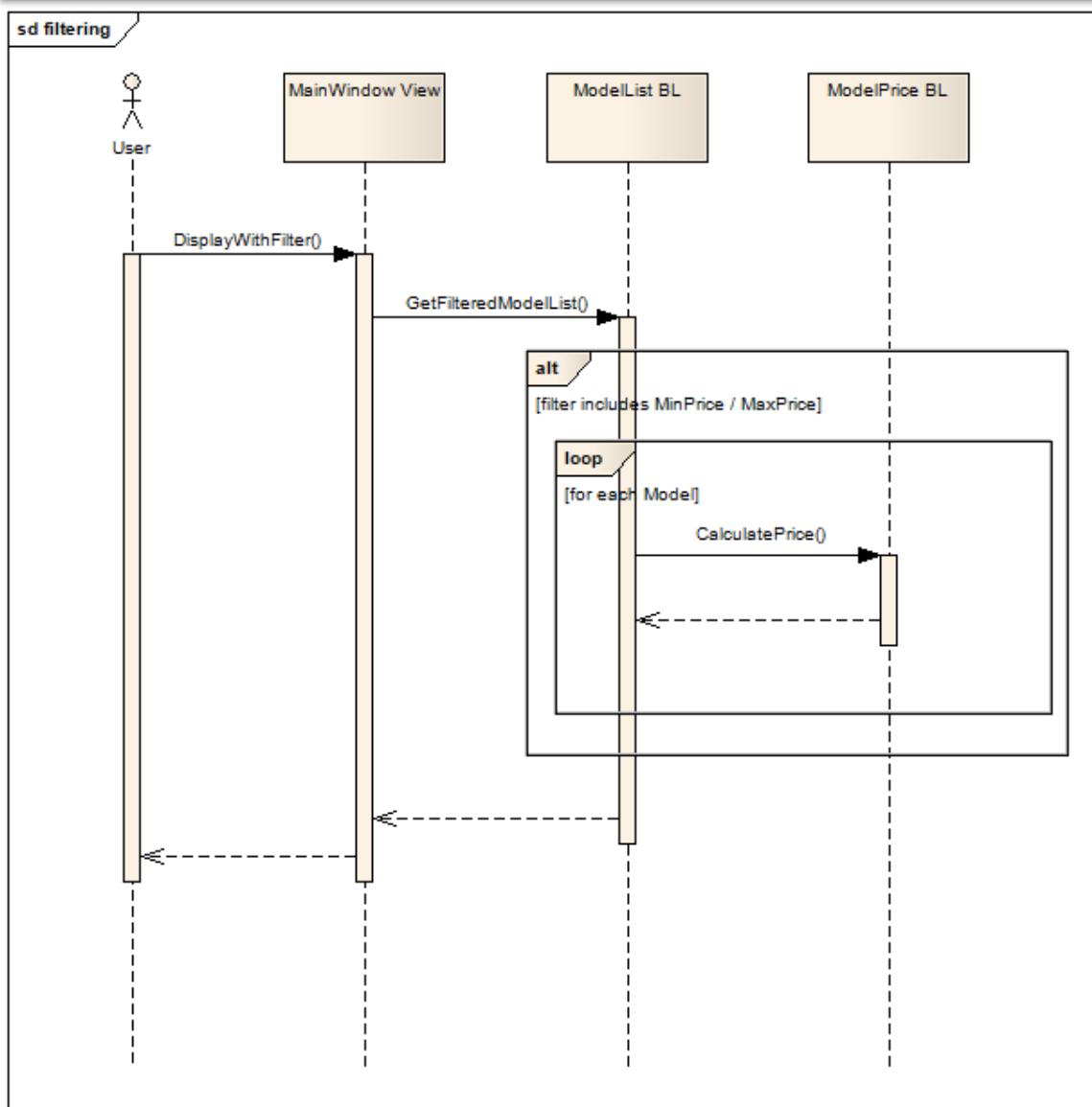
21



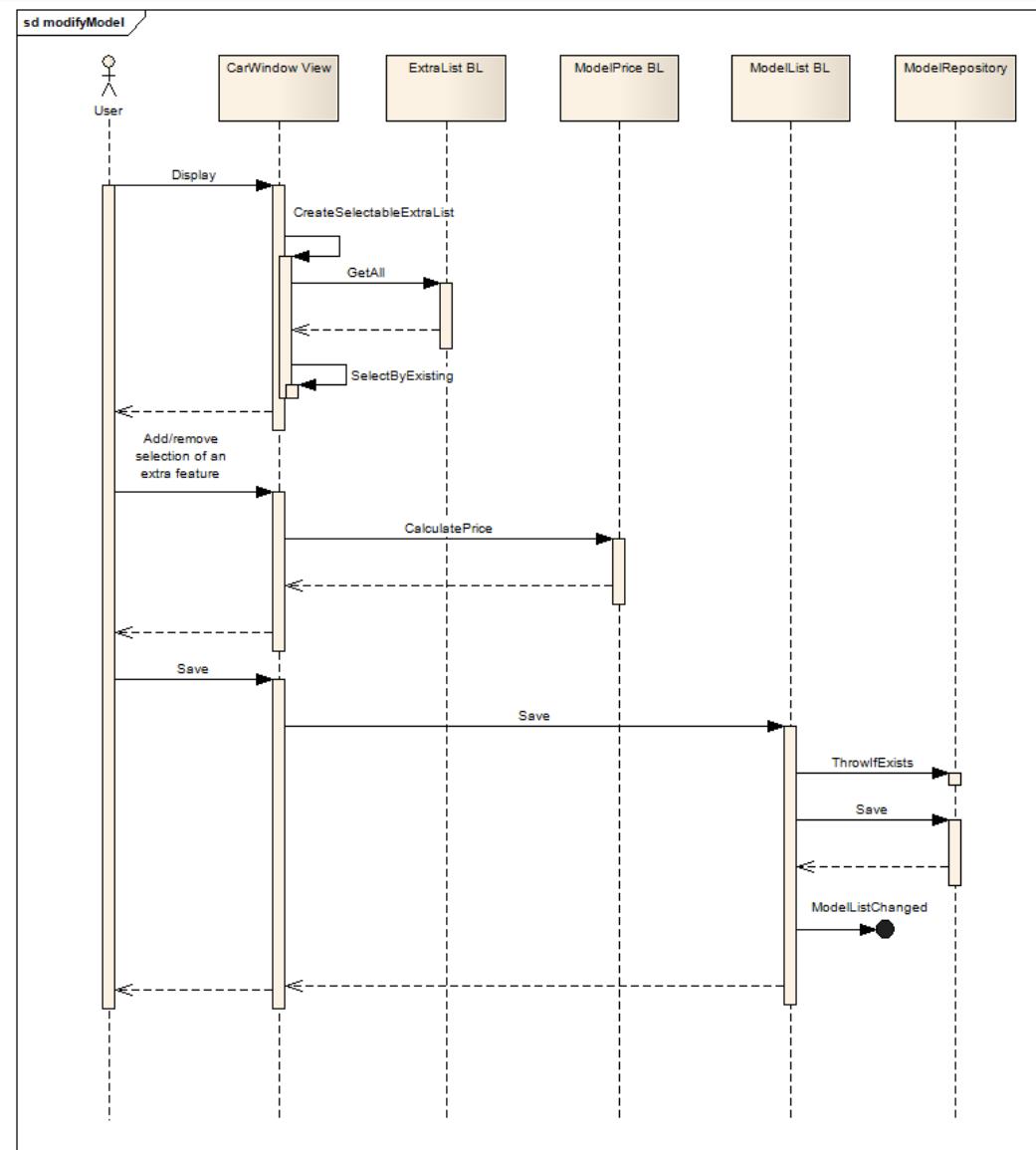
# Sequence diagram



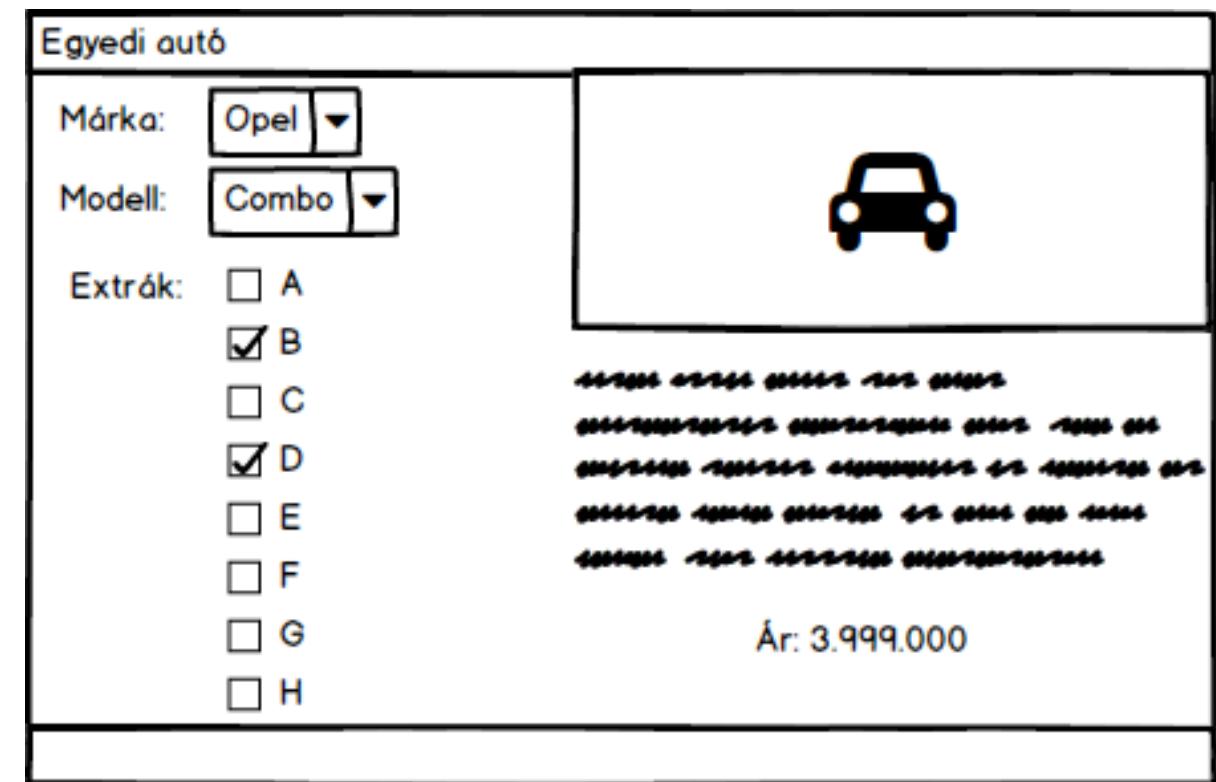
# Sequence diagram



# Sequence diagram



- UML létrehozásakor nem gondoltak arra, hogy a megjelenítés is a modell része lehet
- Legelső prototípus, amit oda tudunk adni a megrendelőnek
- Létrehozása
  - Fejlesztőeszközzel → pl. WPF-ben XAML összerakása adatkötés nélkül
  - Wireframe eszközzel
  - Rajzprogrammal
- Folyamatok előrejelzése kötelező
- Melyik ablakból hogyan és miként lehet átjutni a következőbe és vissza



- **Viselkedési diagramok (behavioral diagram)**

- Használati eset diagram (use-case diagram)
- Aktivitás diagram (activity diagram)
- Állapotgép diagram (state machine diagram)
- Interakciós diagram (interaction diagram)
  - Kommunikációs diagram (communication diagram)
  - Időzítés diagram (timing diagram)
  - Szekvencia diagram (sequence diagram)

- **Kinézet terv (wireframe) ← nem UML**

Előzetes tervezés fázis

- **Strukturális diagramok (structure diagram)**

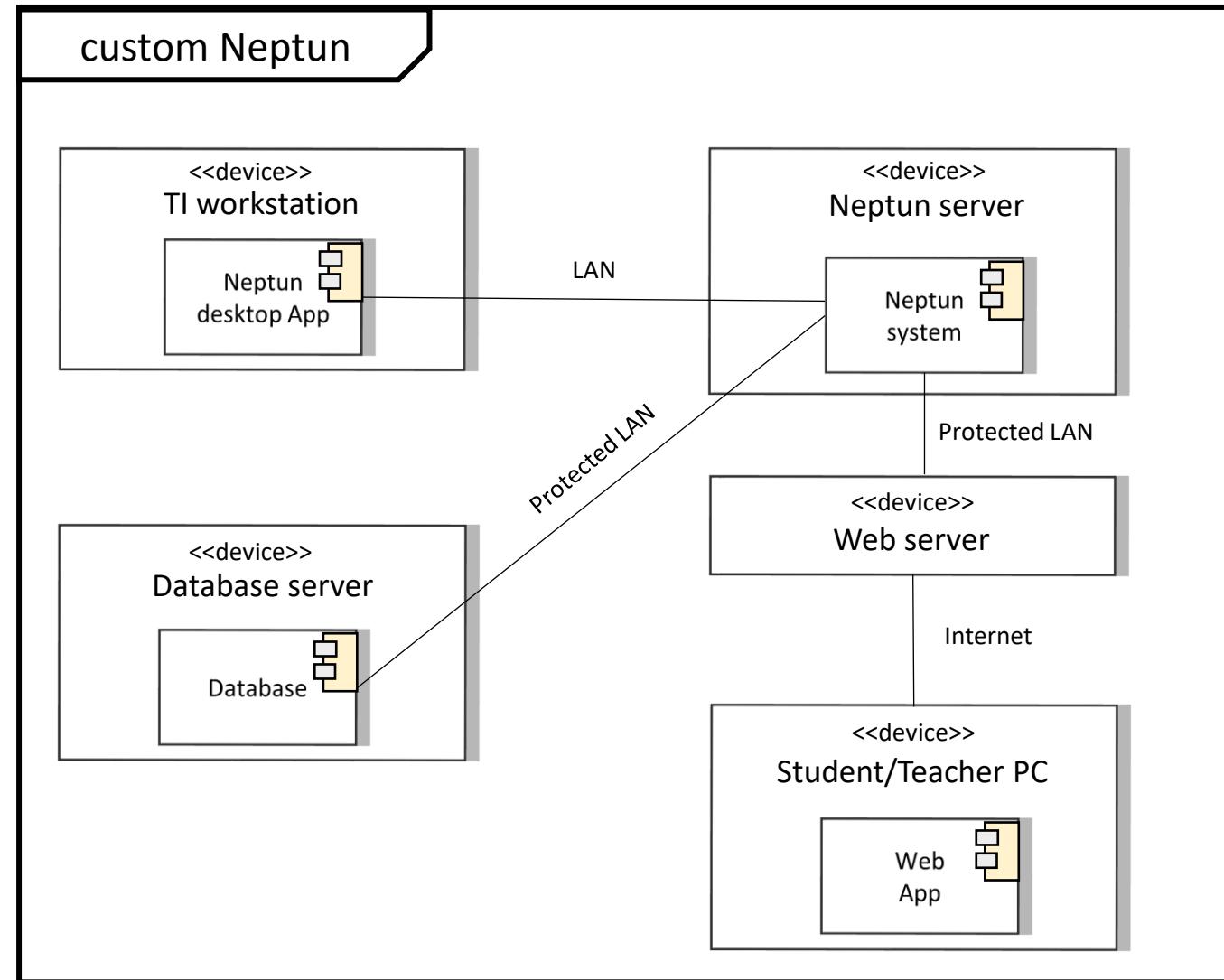
- Osztály diagram (class diagram)
- Komponens diagram (component diagram)
- Objektum diagram (object diagram)
- Kompozit-struktúra diagram (composite structure diagram)
- Telepítési diagram (deployment diagram)
- Csomag diagram (package diagram)

Részletes tervezés /  
Megvalósítás fázis

# Deployment diagram

27

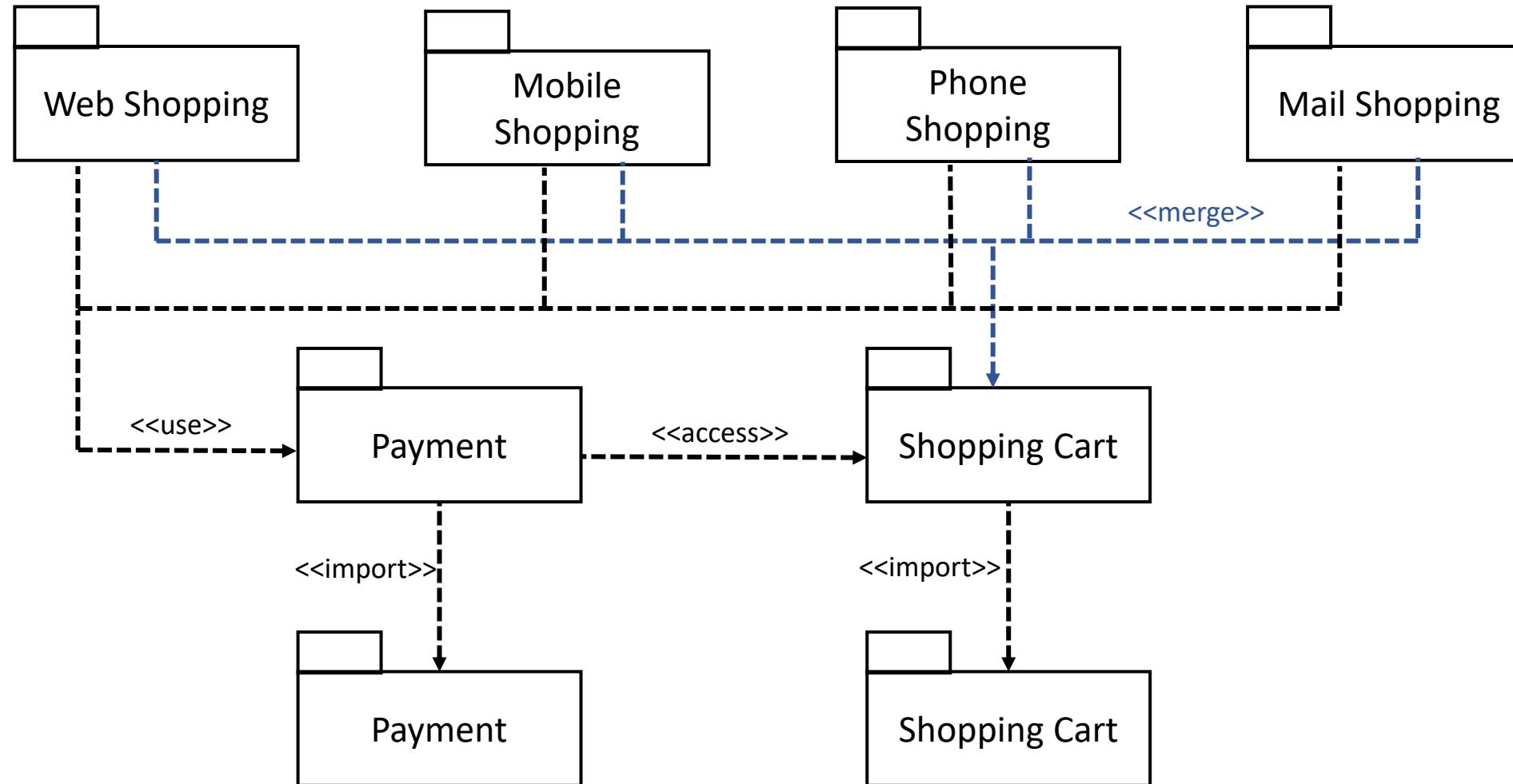
- A rendszer futásának körülményeit/elemeit bemutató diagram
- A működtető elemek lehetnek
  - Számítógépek/kiszolgálók
  - Hálózati csomópontok
  - Egyéb környezetek (VM, konténer, stb.)
- Akár a fejlesztési fázis első diagramja is lehet
  - Ha a környezet már készen van (új szoftvert kell írni meglévő környezetre)
  - De alapvetően új rendszernél a részletes tervezés során használjuk



- Component vs Package diagram
  - Szinte ugyanaz
- Egy lehetséges megközelítés
  - **Package:** felépítés, tartalmazás, függőségek
    - Amiket csomagolni lehet (pl. egy DLL-be)
  - **Component:** kapcsolódások, interfészek
    - Helyi vagy távoli alrendszeren is
- Vagyis a component diagram egy szélesebb leírása, külső függőségeket is tartalmazza

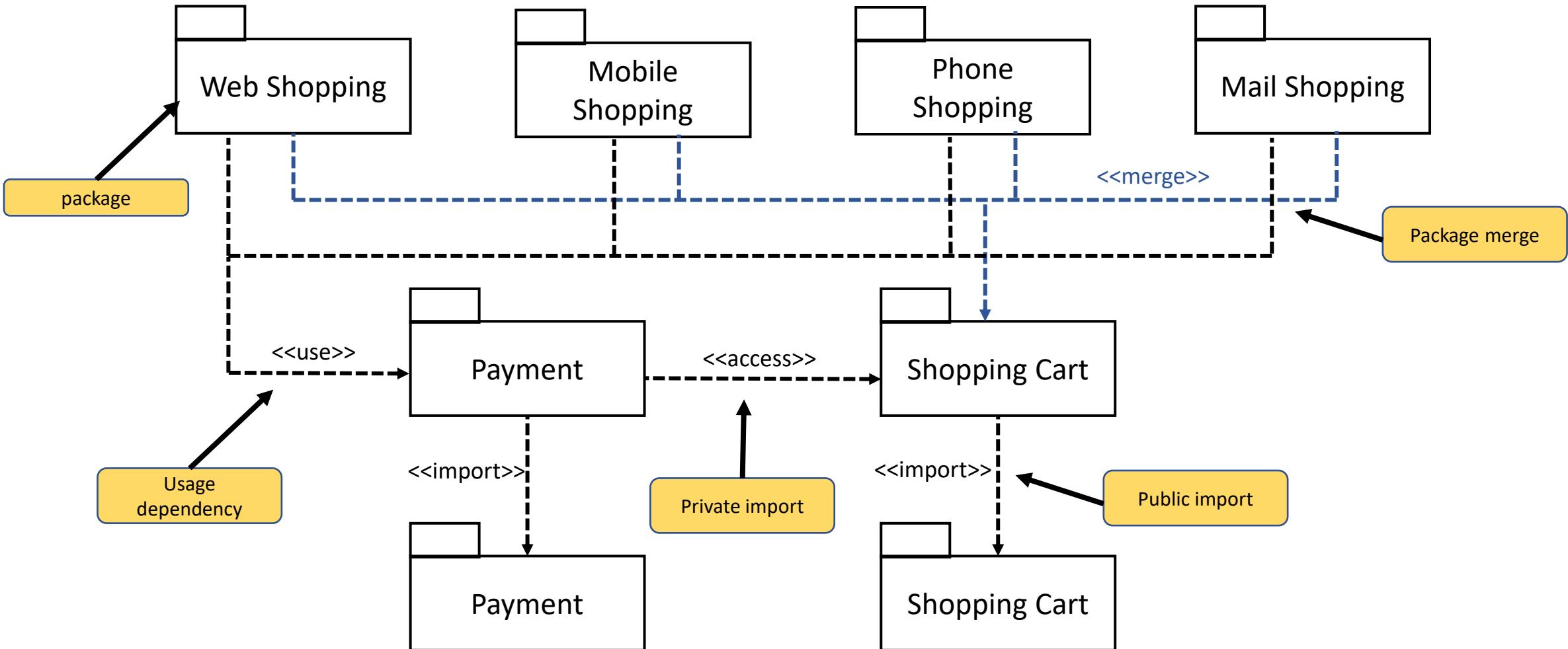
# Package diagram

29



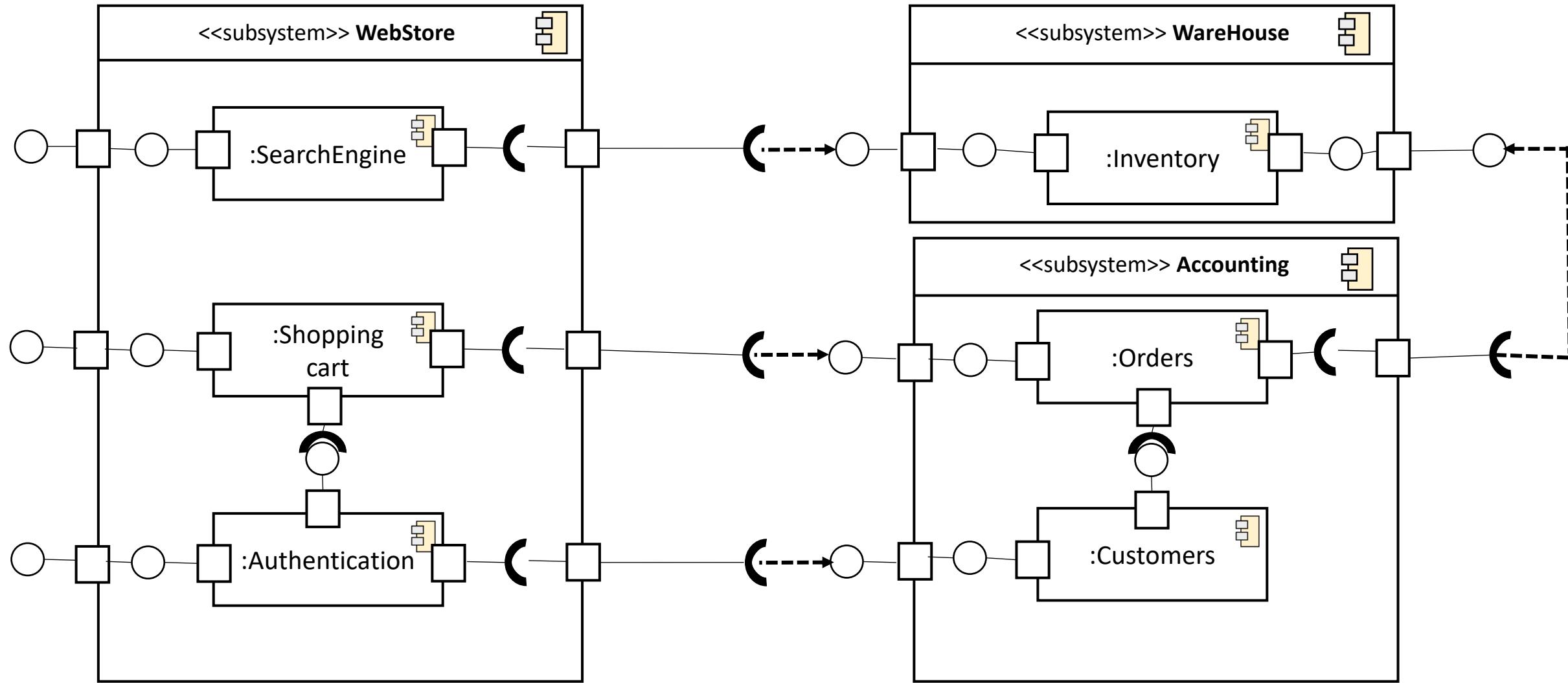
# Package diagram

30



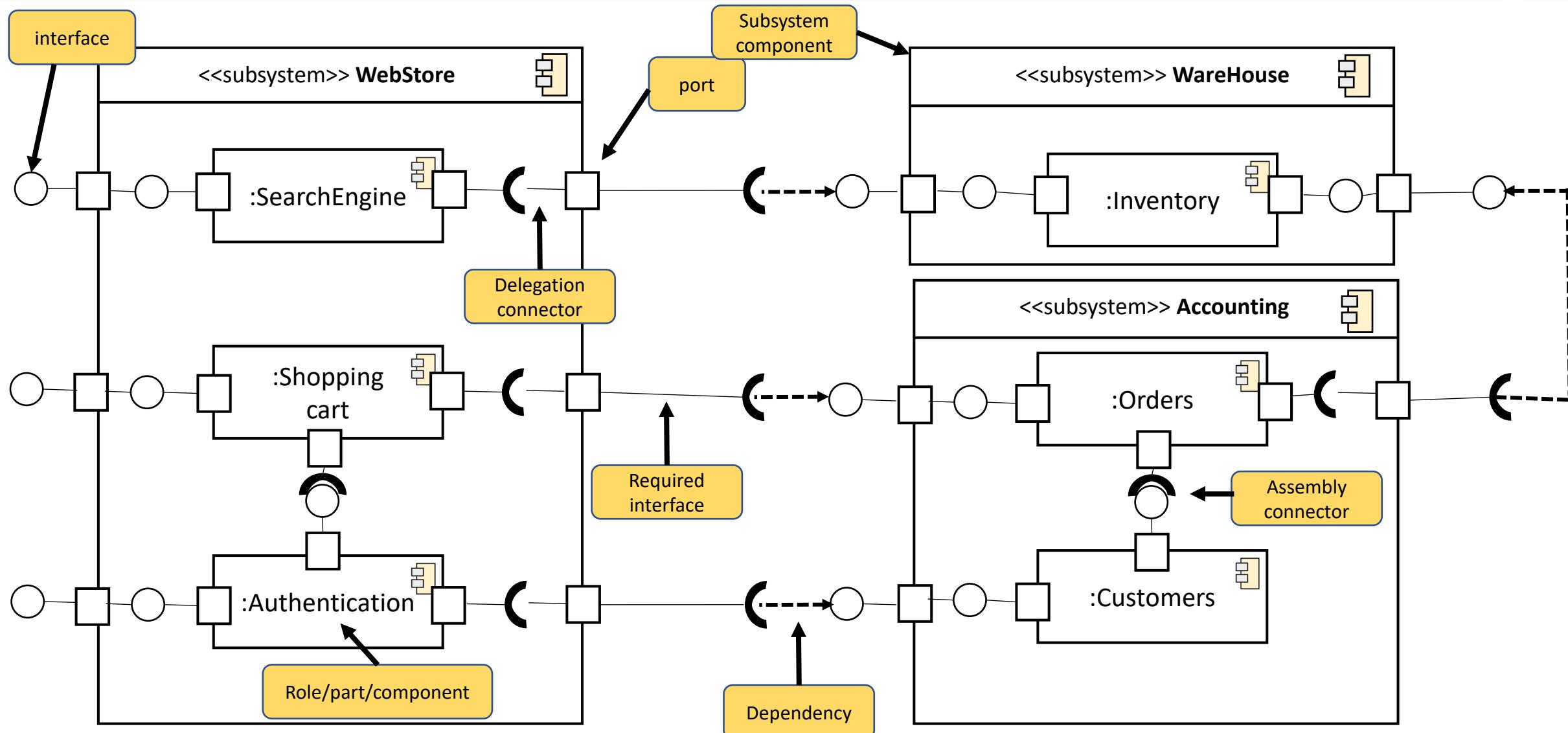
# Component diagram

31



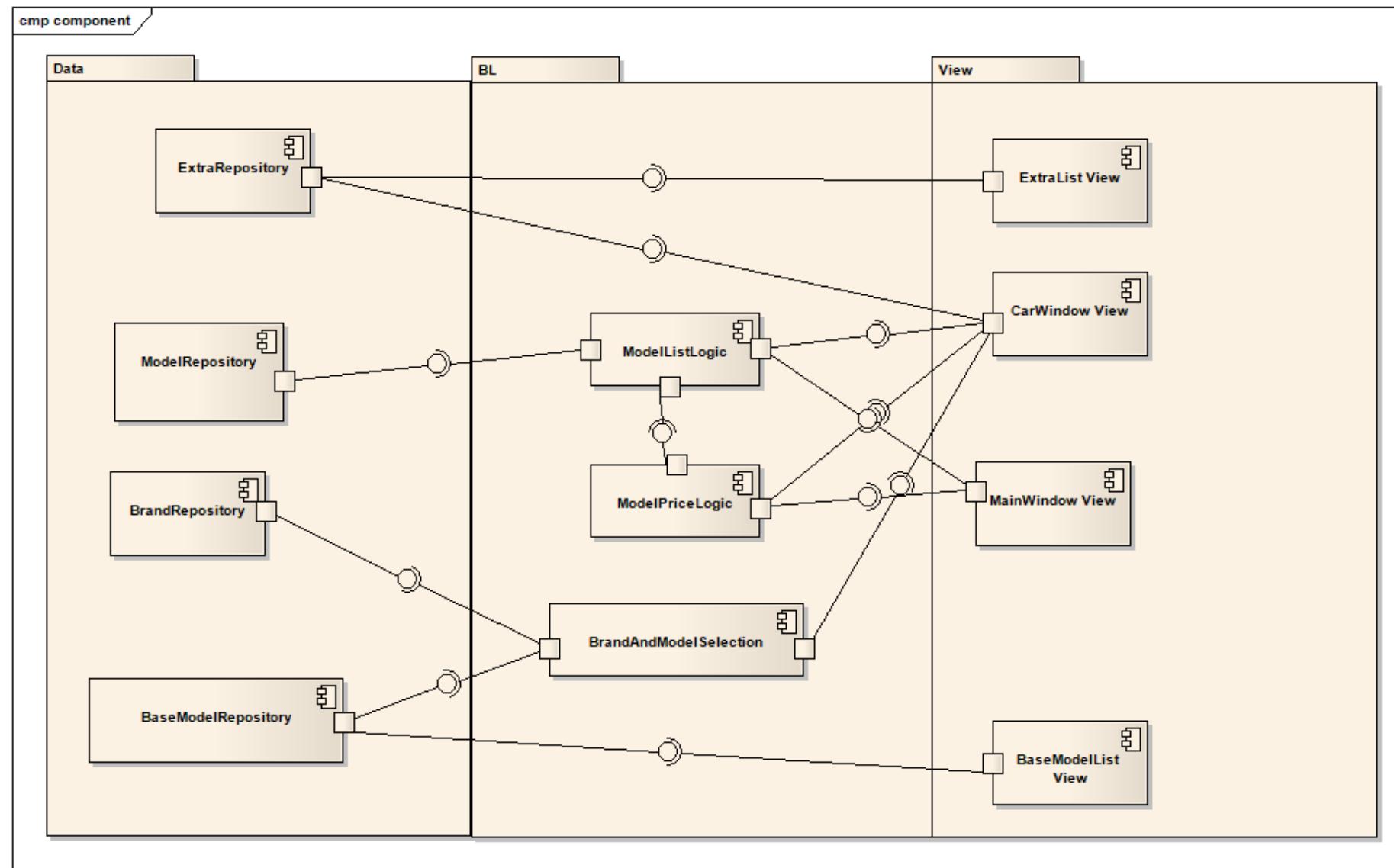
# Component diagram

32



# Component diagram

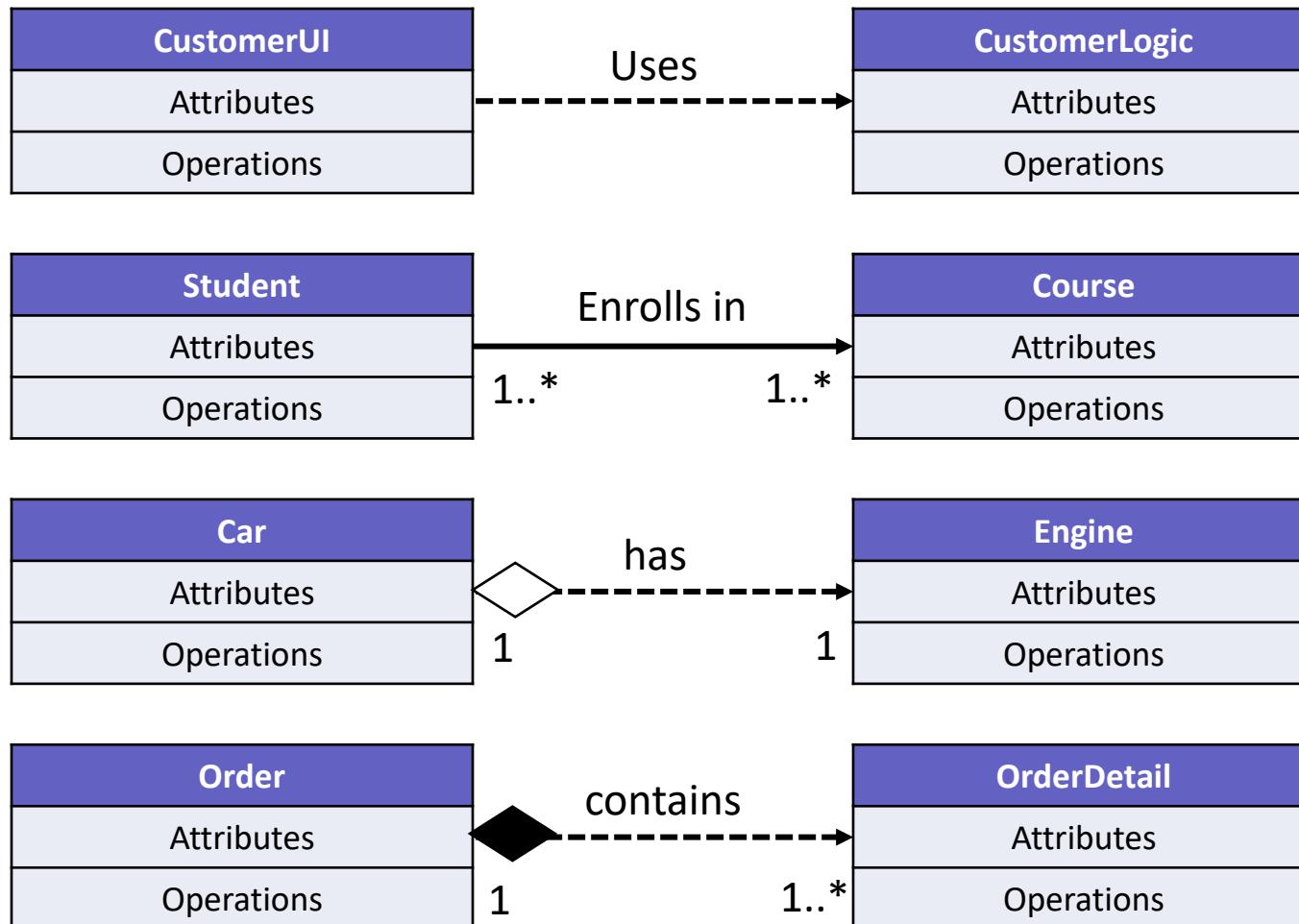
33



- Osztályok vagy konkrét objektumok ábrázolása
  - Adattagok, metódusok
- Mikor készül el?
  - Korai tervezéskor még nem kell
    - Package diagram interfészei elegek
  - Pregroom meeting
    - Architect feladata is lehet akár

# OO kapcsolatok

35



Dependency/típus  
közvetett/közvetlen használata

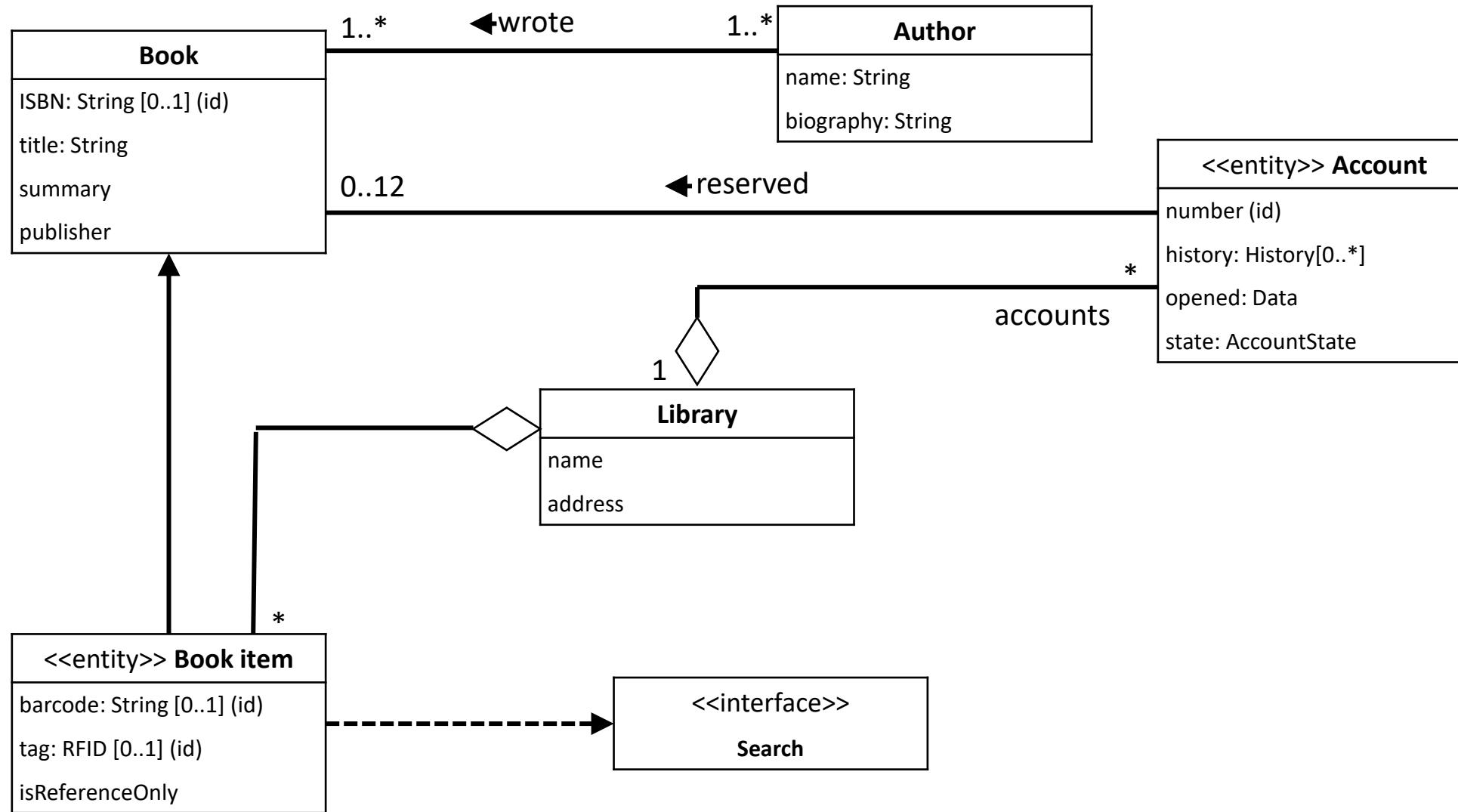
Asszociáció  
teljesen különálló objektumok

Aggregáció  
tartalmazás, de különálló  
élettartam

Kompozíció  
Tartalmazás, függő élettartam

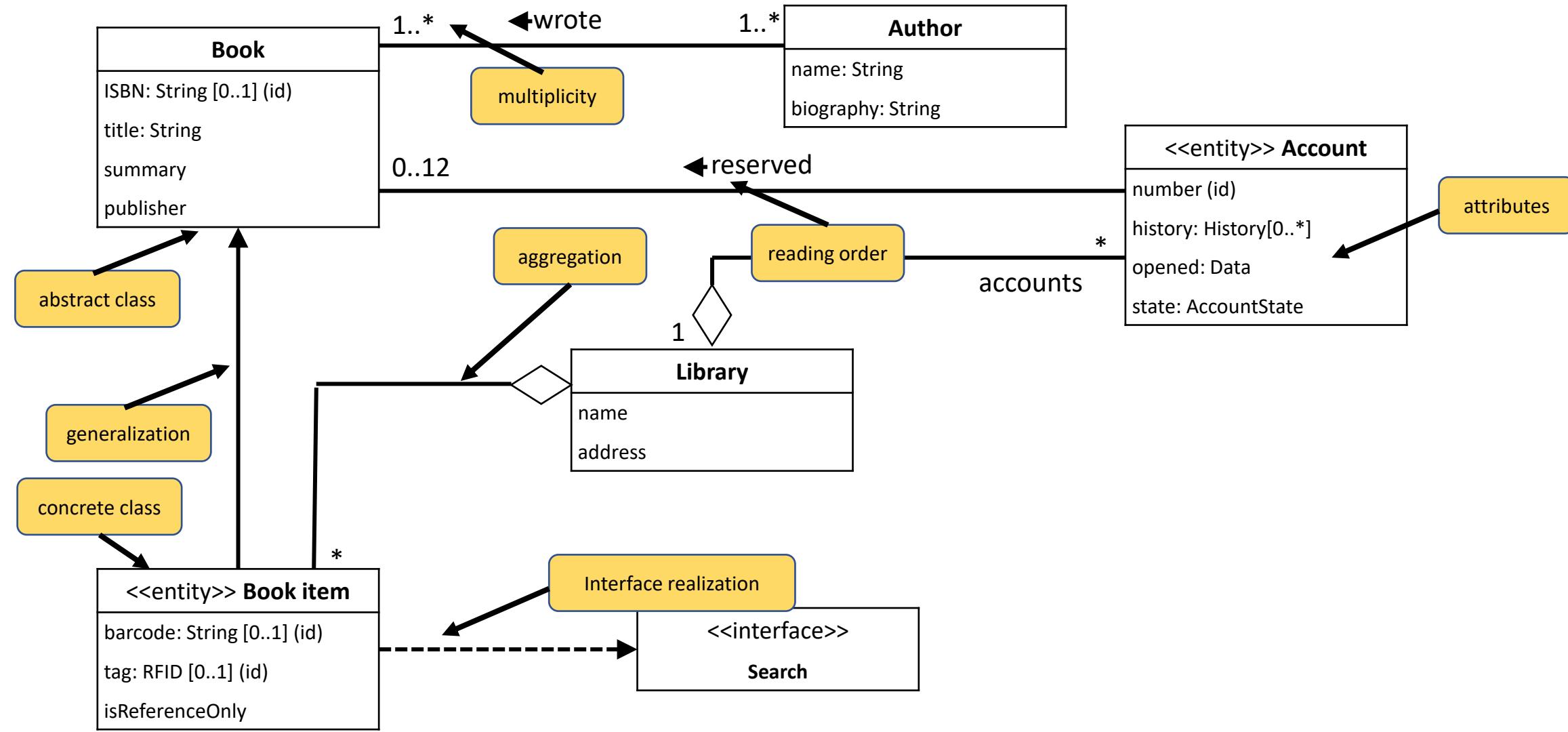
# Class diagram

36



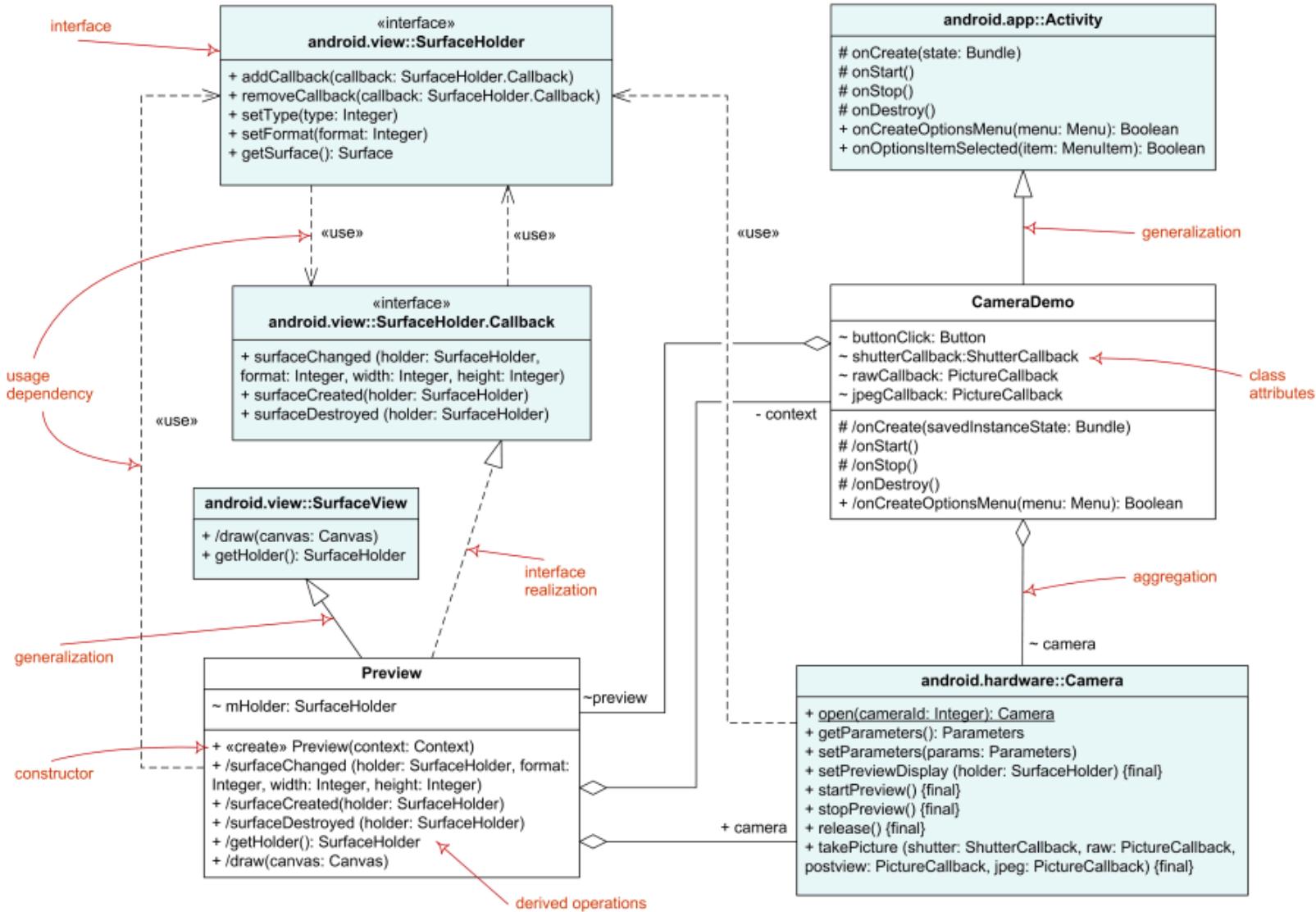
# Class diagram

37



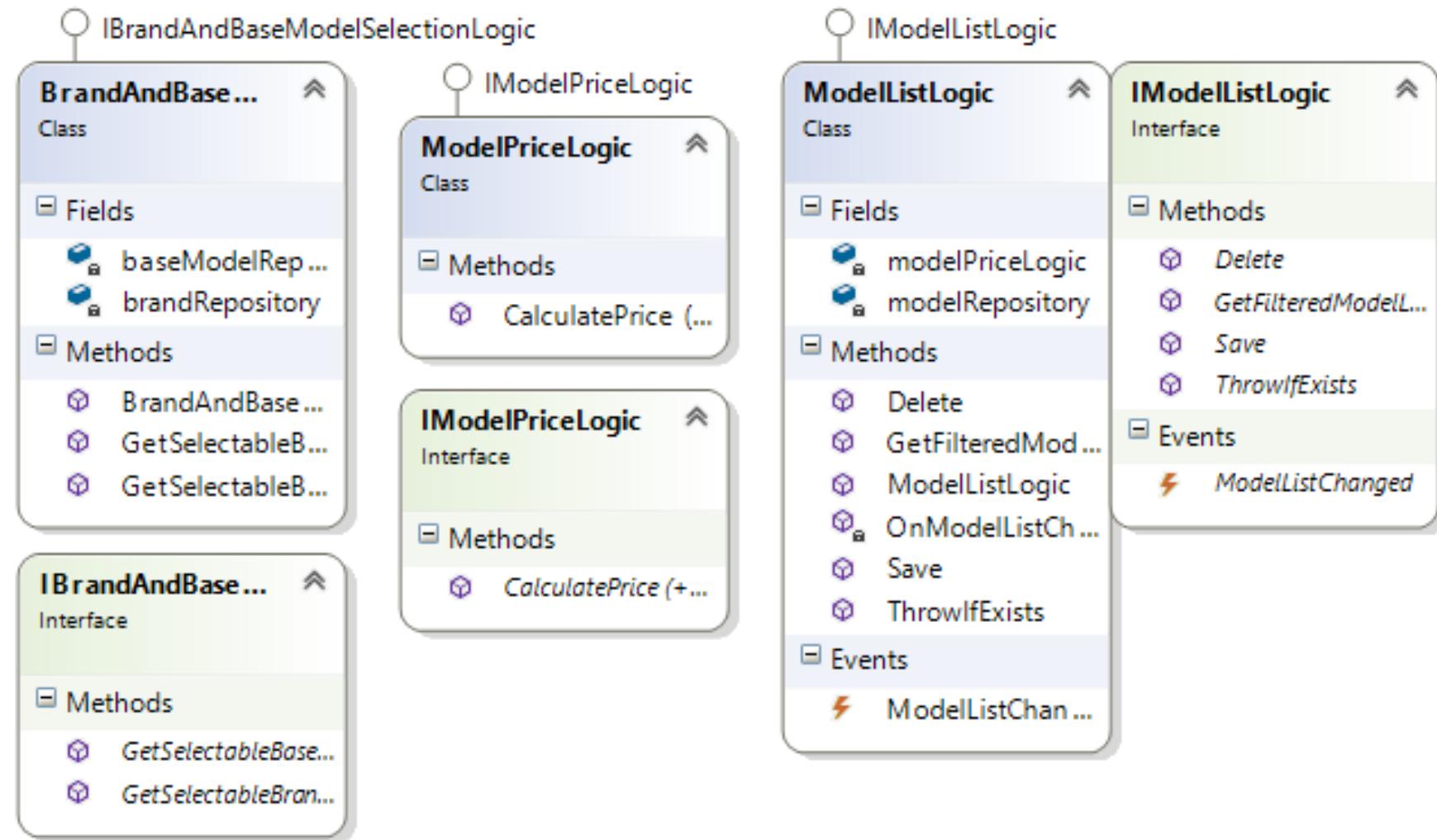
# Class diagram

38

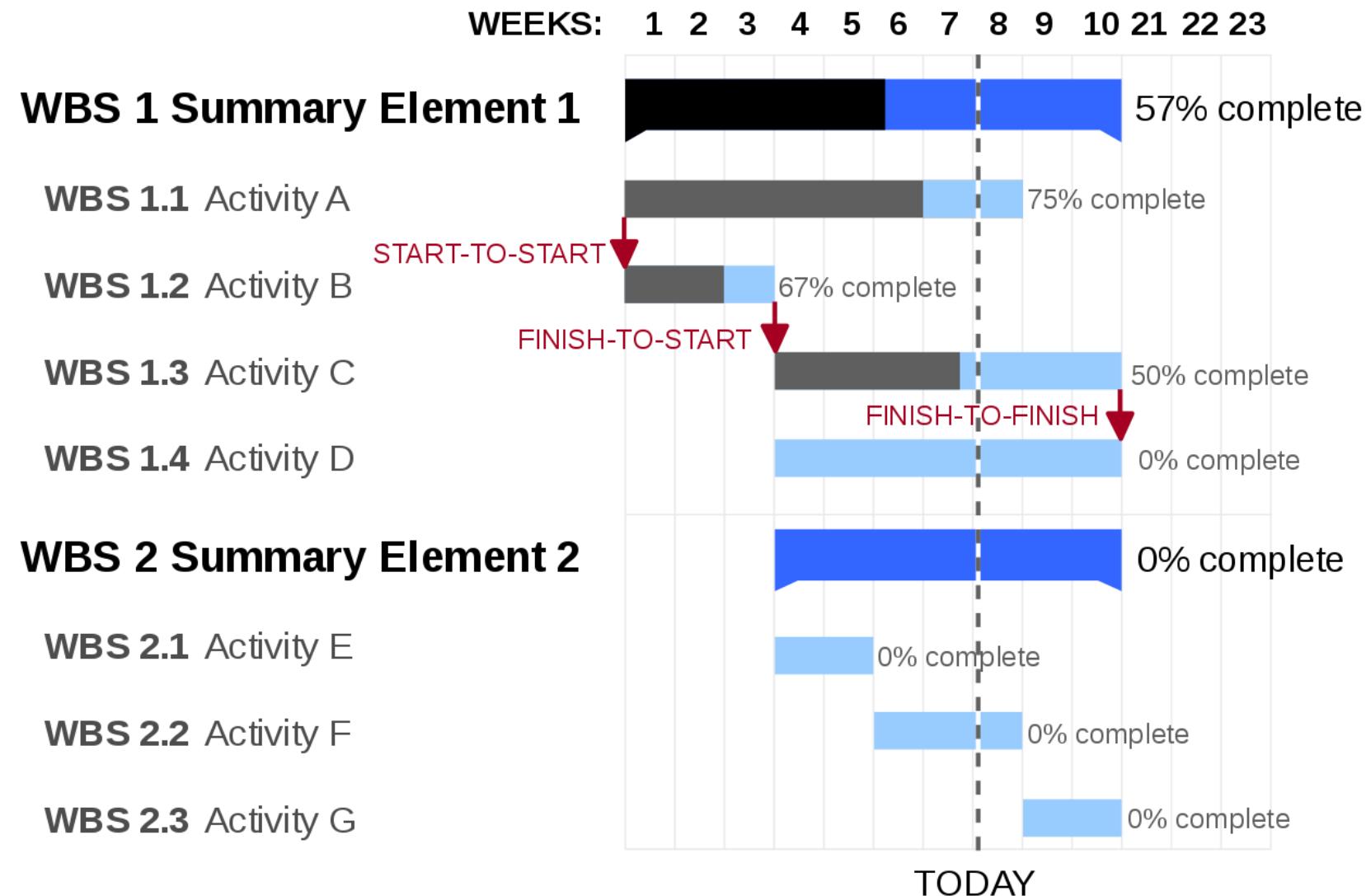


# Visual Studio Class diagram

39



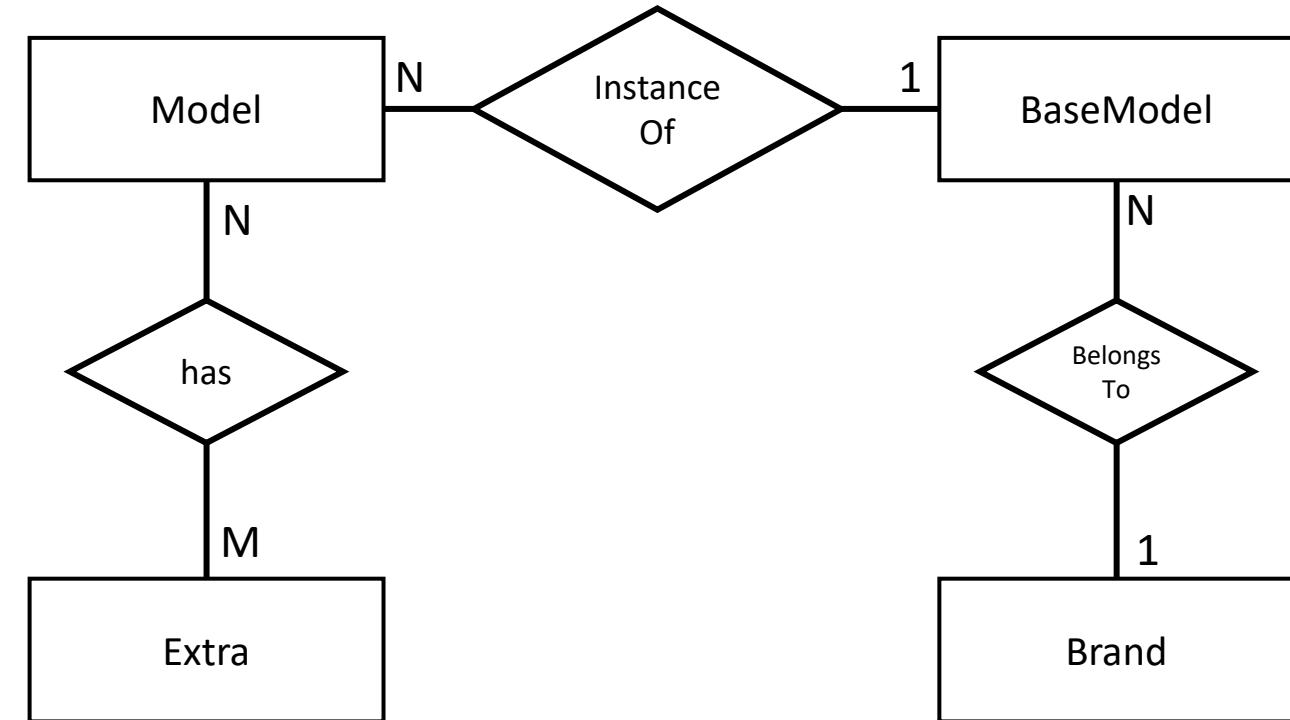
- Időterv készítése
- Egymástól függő folyamatok ábrázolása
- Nem UML diagram



# Entity-Relation diagram

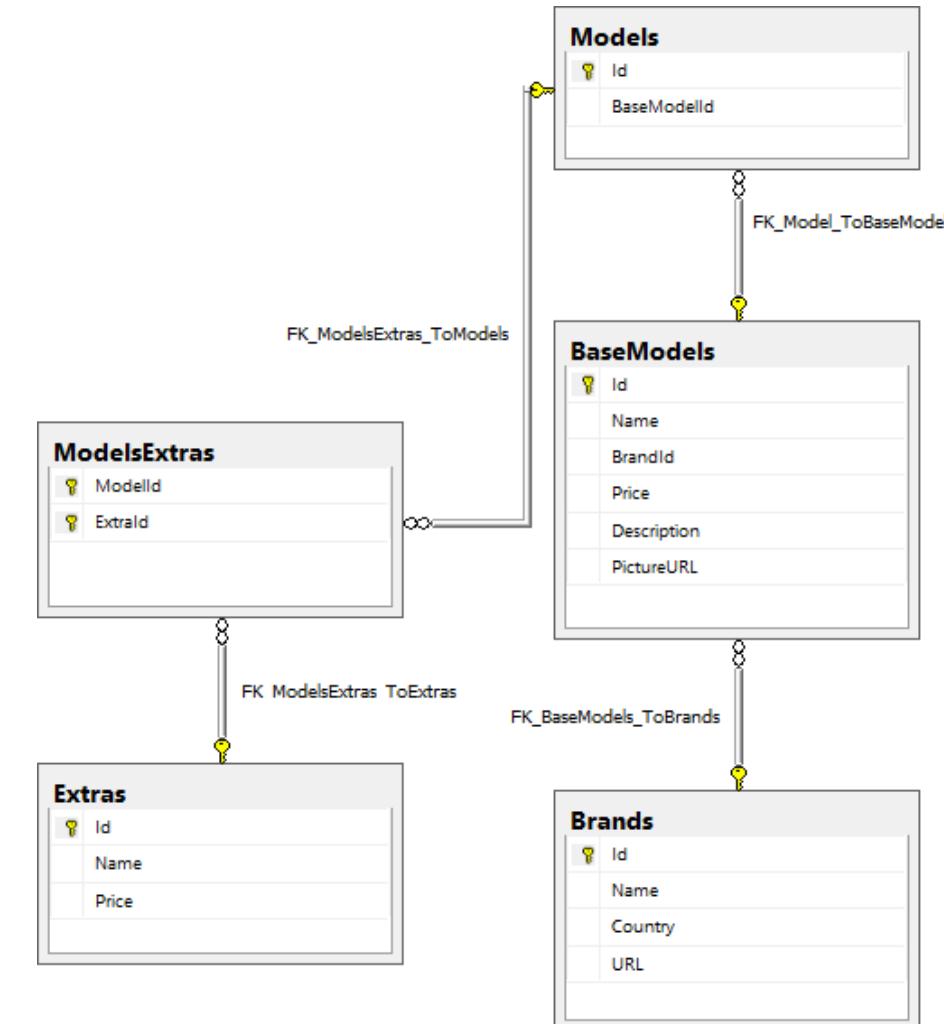
41

- Nem UML diagram
- Adatmodellezéskor használjuk
- Egyedek és köztük lévő kapcsolatok tervezése
  - Majd normalizálás
  - Majd adatbázis készítése



# Table structure diagram

- Szintén nem UML diagram
- Adatbázis táblák szemléltetése
- Elsődleges és idegen kulcsok ábrázolása
- Gyakorlatilag minden szoftverfejlesztési projekthez kell



## 1. Deployment diagrams

- A rendszer hogyan és milyen körülmények között lesz használva

## 2. Behavioral diagrams

- Milyen funkciókat kell a rendszernek tudnia?
- Use-case + Activity + Wireframes
- Megrendelővel közös tervezés

## 3. Structural diagrams

- A működést milyen modulokkal, milyen felbontással lehet megoldani
- Component + Sequence → Class diagrams
- Entity-relations diagram → DB tábla struktúrák

## 4. Időtervezés → Gantt diagram

## 5. Implementálás

# Köszönöm a figyelmet!

Kérdés esetén e-mailben szívesen állok rendelkezésre.



# Szoftvertechnológia



ÓBUDAI EGYETEM  
NEUMANN JÁNOS INFORMATIKAI KAR

# **MODUL 5**

**SOLID elvek ismétlése**  
**Architekturális tervezési minták összefoglalása**  
**GOF**  
**Létrehozási tervezési minták**

# Gyors szoftverfejlesztés

- Kizárolag csapatmunkában
- Multibranch GIT környezetben
- Projektmenedzsment tool-ok támogatásával
- Újrahasznosított komponensekkel
- Frameworkök használatával
- Folyamatos teszteléssel
- Folyamatos integrációval
- Jól bevált, profik által javasolt kód mintákkal
- Folyamatos vevői véleményeztetéssel
- Fenntartható, moduláris kód írásával

- SZTF I-II. tárgyak féléves feladataiban általában nekiugrottunk a kódnak és megírtuk valahogyan
- Törekedtünk arra, hogy használjuk az OOP-t
- Nagyobb rendszereknél ez nem működik, mert
  - Túl komplex az alkalmazás
  - Átláthatatlan lesz
  - Esélytelen a későbbi kiegészítés
  - Platformfüggő (konzol → WPF átalakítás gyakorlatilag lehetetlen)
- Megoldás: **tervezési minták (desing patterns)**
- Mi a fontosabb?
  - Kódhossz, megírási idő, futásidő **VS** olvashatóság, újrafelhasználhatóság, karbantarthatóság

# Tervezési minták

- A szoftverfejlesztésben ritkán van „új a nap alatt”
- Általában belefutunk olyan feladatokba, amelyekre már valaki rutinosabb fejlesztő előállt egy nagyon szép, nagyon jól szervezett kóddal, ami bővíthető, moduláris, stb.
- Ezeket a jól bevált sémákat nevezzük tervezési mintának
- Olvasnivaló a témaban
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software
  - Horváth Rudolf: Common Design Patterns
  - Aniruddha Chakrabarti: Design Patterns (GoF) in .NET
  - <http://dofactory.net/net/design-patterns>
  - Michael Feathers: Working Effectively with Legacy Code
  - Joshua Kerievsky: Refactoring to Patterns
  - Martin Fowler: Refactoring (a.k.a. The Refactoring Bible)

- **Single Responsibility**

- minden osztály egy dologért legyen felelős (és azt jól lássa el)
- Ha nem követjük, akkor
  - Spaghetti kód, átláthatatlanság
  - Gigászi méretű God object-ek
  - mindenért felelős alkalmazások és szolgáltatások
  - Alkalmazás szinten egyre kevésbé követik (pl. winamp, chrome, systemd, moodle)

- **Open/Closed Principle**

- Egy osztály legyen nyitott a bővítésre és zárt a módosításra (nem írhatunk bele, de származtathatunk tőle)
- Ha nem követjük, akkor
  - Átláthatatlan, lekövethetetlen osztályhierarchiák, amelyek nem bővíthetők
  - Tünet: leszármazott megírásakor módosítanunk kell az ősosztályt is (TILOS, frameworknél pedig lehetetlen)
  - Tünet: egy kis funkció hozzáadásakor több osztályt kell hozzáadni ugyanabban a hierarchiában

- **Liskov substitutable**

- Ősosztály helyett utódpéldány legyen minden használható
- Compiler supported, hiszen OOP elv (polimorfizmus)
- Ha egy kliensosztály eddig X osztállyal dolgozott, akkor tudnia kell X leszármazottjával is dolgozni
- Ha a kliensosztály úgy tudja, hogy X [5..10] intervallumon értékeket fogad, akkor Y osztály tágabb intervallumot fogadhat, szűkebbet nem
- Ha a kliensosztály úgy tudja, hogy X [5..10] intervallumon értékeket biztosít, akkor Y osztály tágabb intervallumot nem adhat vissza, szűkebbet igen

- **Interface segregation**

- Sok kis interfészt használunk egy hatalmas minden előíró interfész helyett
- Ha nem követjük, akkor
  - Tünet: Egy osztályt létrehozunk valamelyen célból, megvalósítjuk az interfész és rengeteg üres, fölösleges metódusunk lesz
  - Tünet: Az interfészhez több implementáló osztály jön létre a kód legkülönbözőbb helyein, más-más részfunkcionalitással

- **Dependency Inversion**

- A függőségeket ne az ōket felhasználó osztály hozza létre
- Várjuk kívülről a példányokat interfészeken keresztül
- Példány megadására több módszer is lehetséges
  - Dependency Injection (ismerjük már)
  - Inversion of Control (IoC) container (ismerjük már: ASP-ben és WPF-ben is használtunk ilyet)
  - Factory tervezési minta (most)
- Ha nem követjük, akkor
  - Egymástól szorosan függő osztályok végtelen láncolata
  - Nem lehet modularizálni és rétegezni
  - Kód újrahasznosítás lehetetlen

- **Egyéb elvek**

- **DRY** = Don't repeat yourself
- **DDD** = Domain Driven Design (a félév vége felé lesz róla szó)

# Architekturális tervezési minták

- Nagyobb alkalmazás alapvető osztályait, működési módját és technológiáját határozza meg
- Három nagyobb architektúra
  - **MVC**: Model-View-Controller
    - JAVA tantárgyon ismerkedtünk meg vele
    - ASP.NET és MVC kötválon mutatjuk be C#-ban
  - **MVVM**: Model-View-ViewModel
    - GUI tantárgyon ismerkedtünk meg vele WPF-ben
    - GUI tantárgyon megnéztük a JS alapjait, de ott nem rétegeztünk (sima JS nem is támogatja)
    - JS keretrendszerek MVVM-ben: Angular, Vue.js (React nem!)
  - **MVVMC v1**: Model-View-ViewModel-Controller
    - ASP.NET és MVC kötválon találkozhatunk vele
  - **MVVMC v2**: Model-View-ViewModel → [API] → Controller
    - HFT tantárgyon megtanultunk API endpointot készíteni (ASP-n bővebben, részletesebben)
    - GUI tantárgyon készítettünk WPF API kienst MVVM-ben és JS API kienst

CONTROLLER

VIEW

MODEL

Logic

Repository

Egyed osztályok

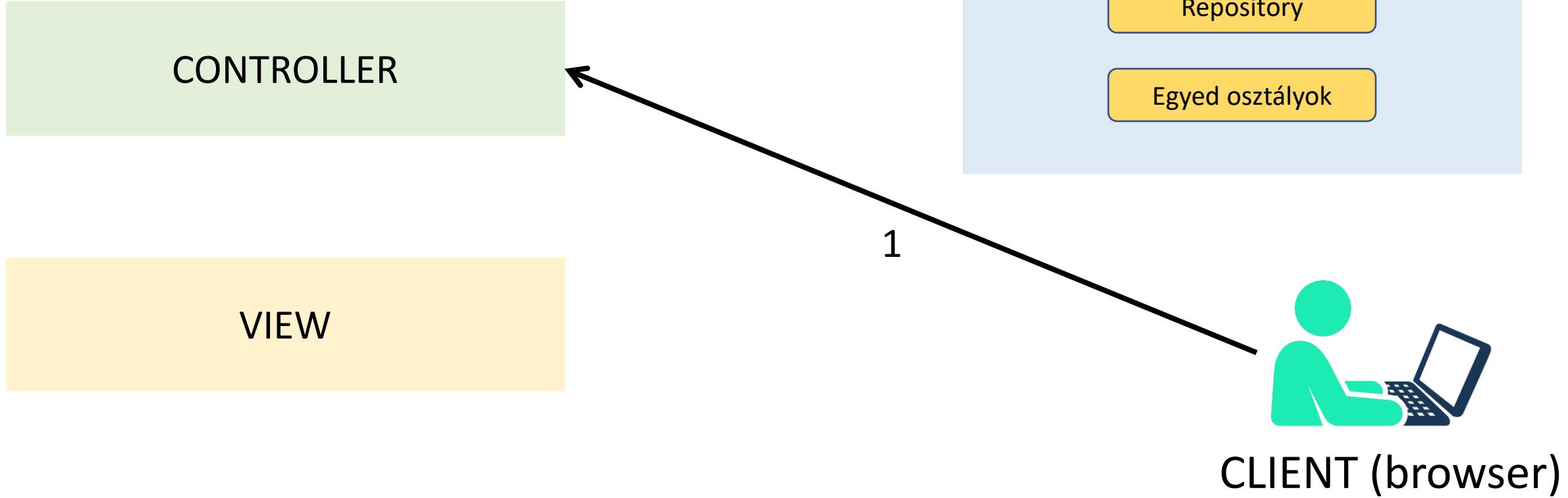


CLIENT (browser)

# MVC workflow

11

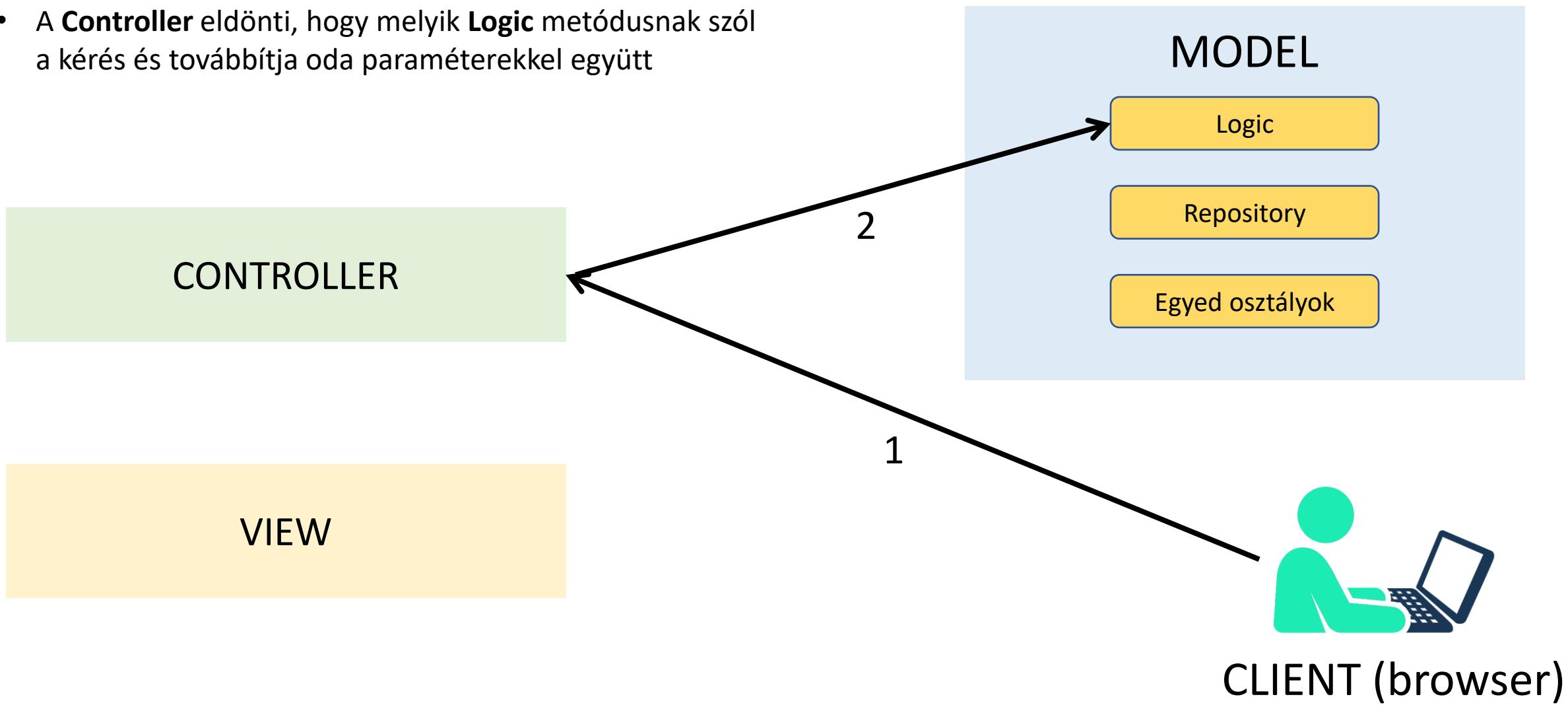
- A felhasználó egy **URL**-re navigál, ez a routing függvényében valamelyik **Controllerhez** kerül



# MVC workflow

12

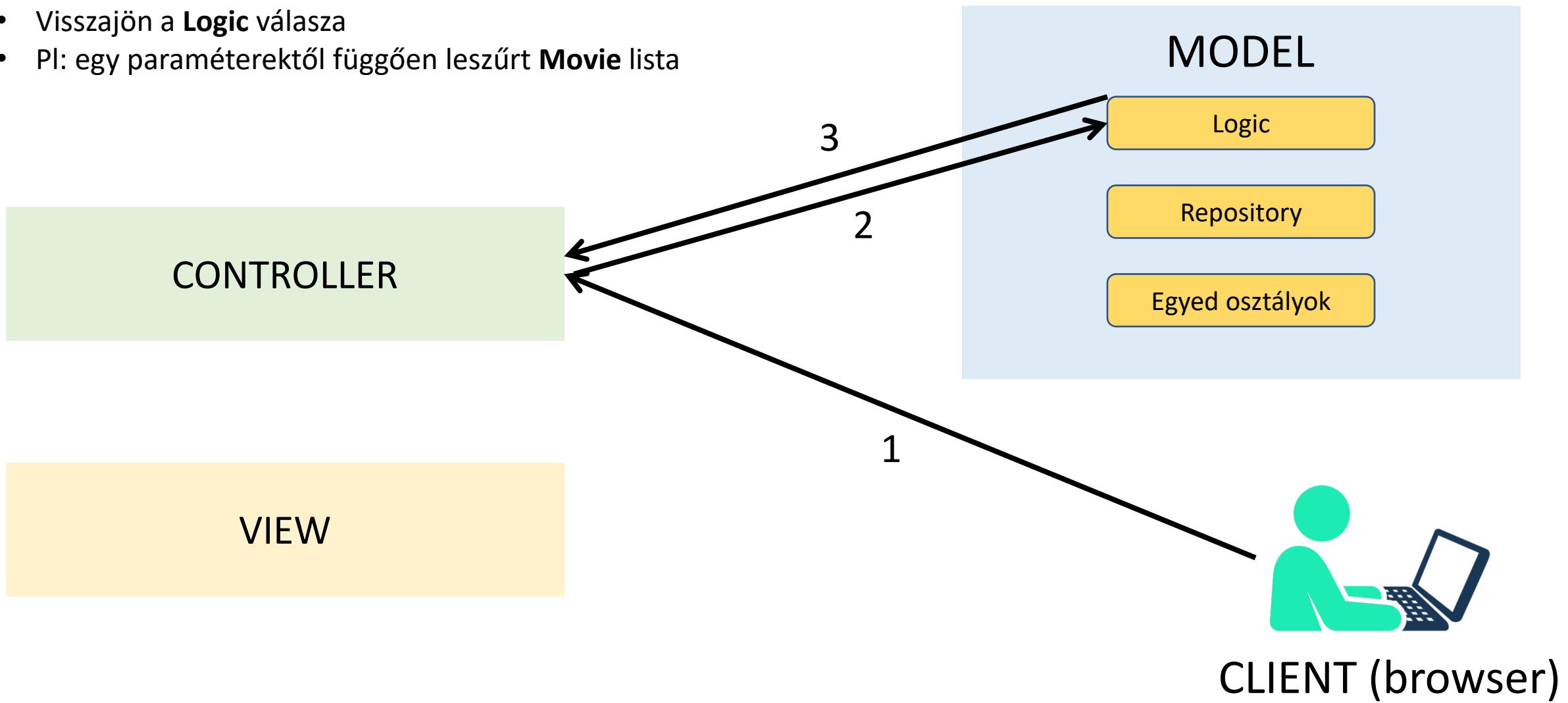
- A **Controller** eldönti, hogy melyik **Logic** metódusnak szól a kérés és továbbítja oda paraméterekkel együtt



# MVC workflow

13

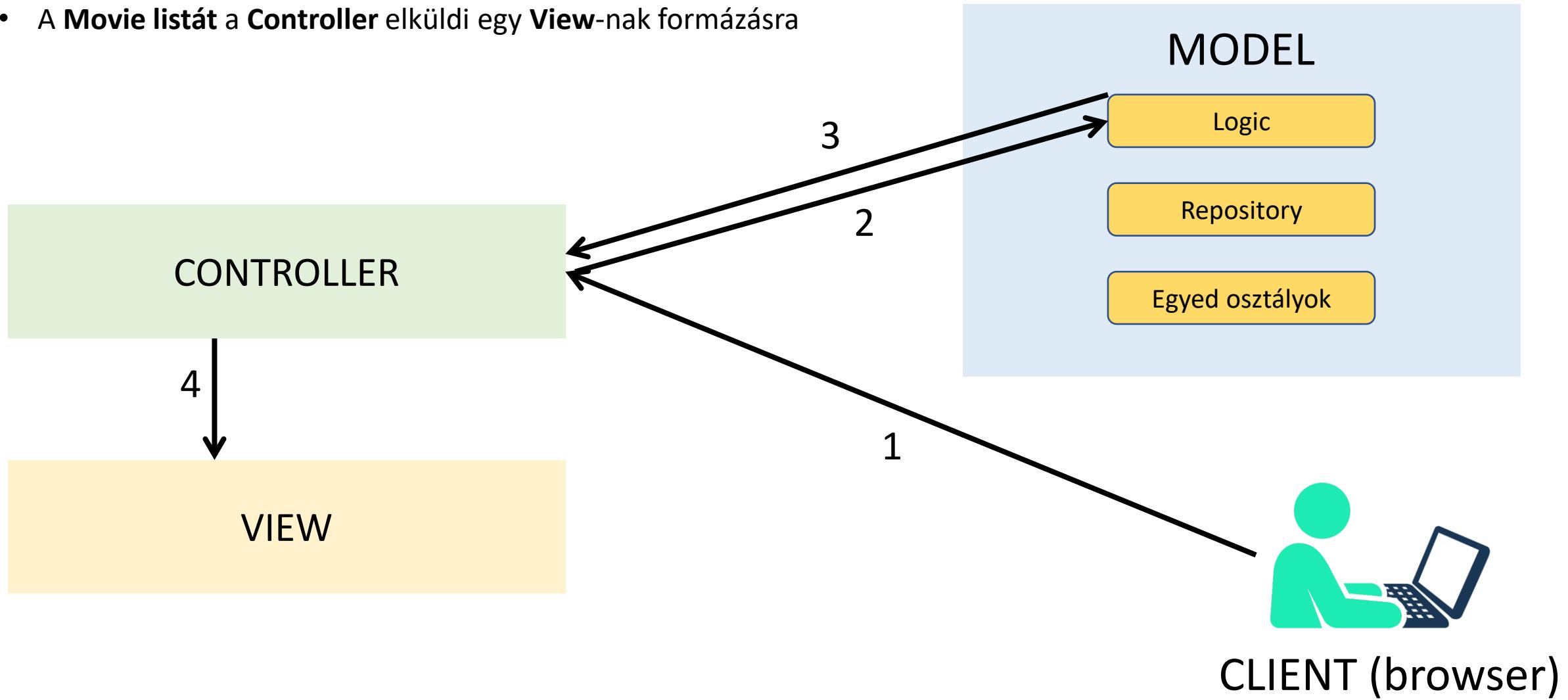
- Visszajön a **Logic** válasza
- Pl: egy paraméterektől függően leszűrt **Movie** lista



# MVC workflow

14

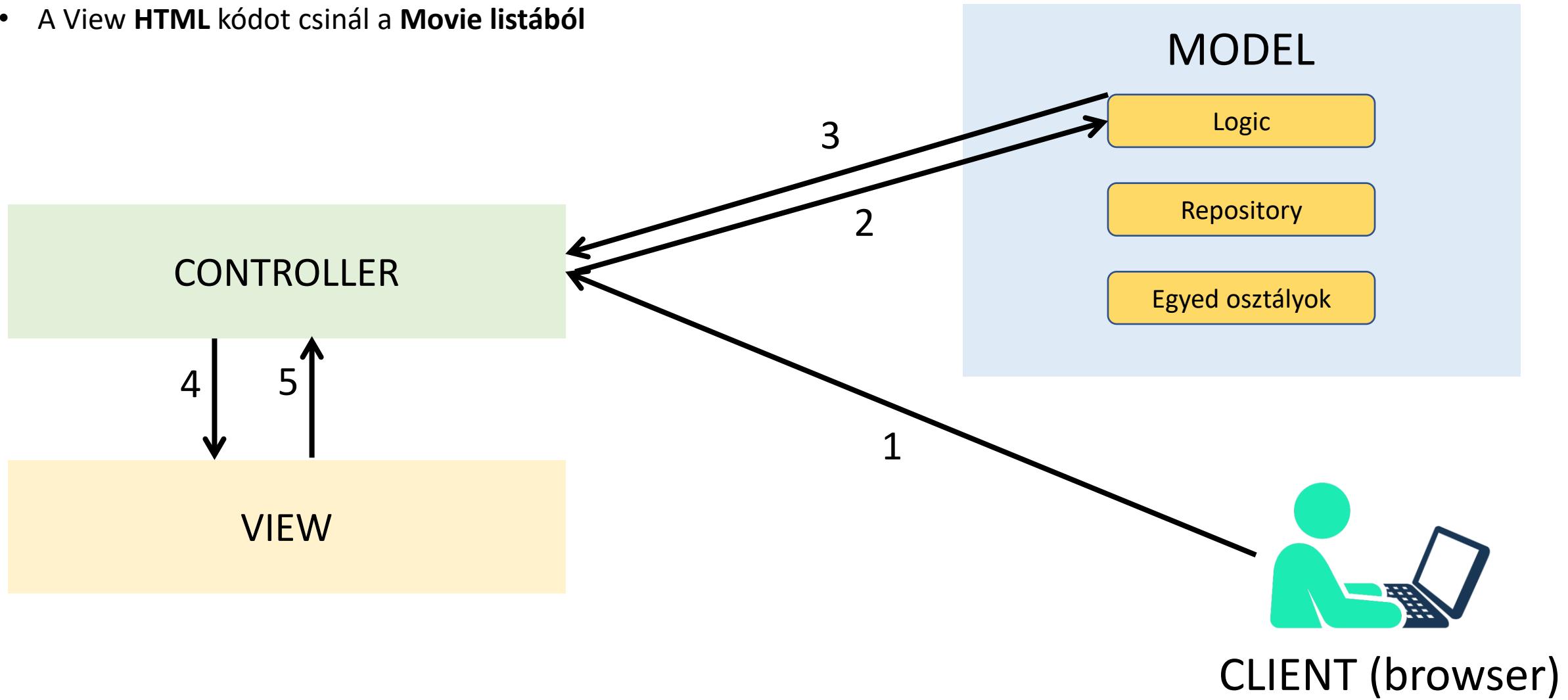
- A Movie listát a Controller elküldi egy View-nak formázásra



# MVC workflow

15

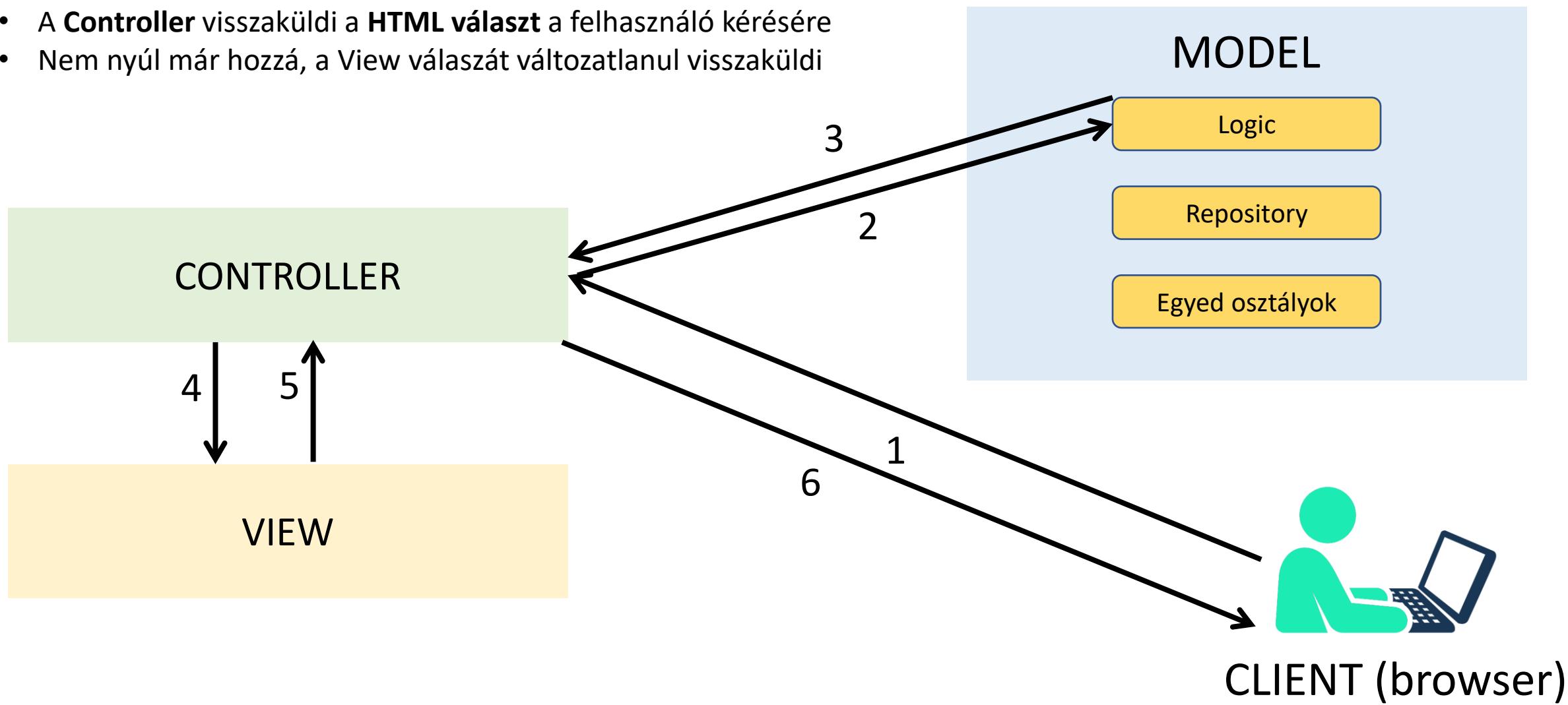
- A View HTML kódot csinál a Movie listából



# MVC workflow

16

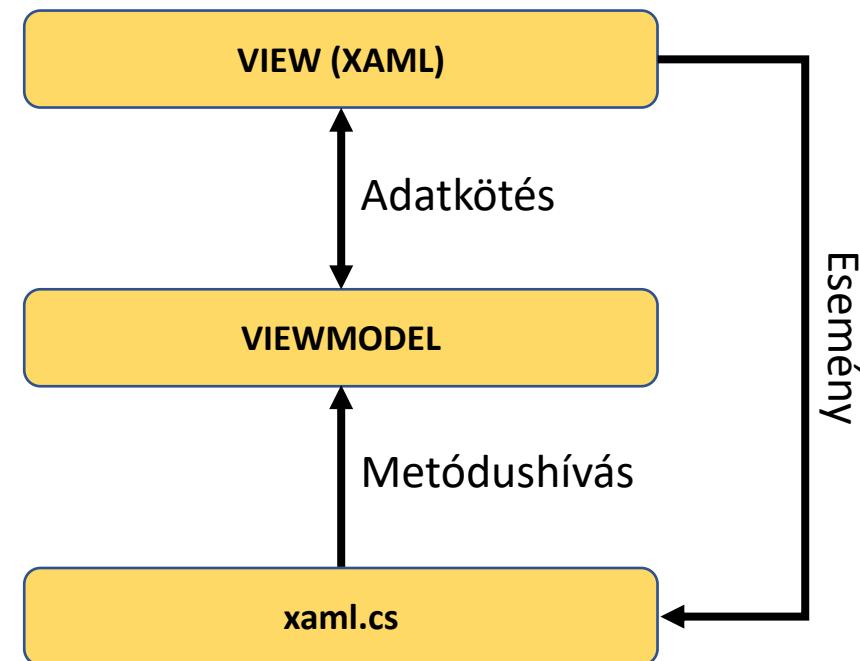
- A **Controller** visszaküldi a **HTML választ** a felhasználó kérésére
- Nem nyúl már hozzá, a View válaszát változatlanul visszaküldi

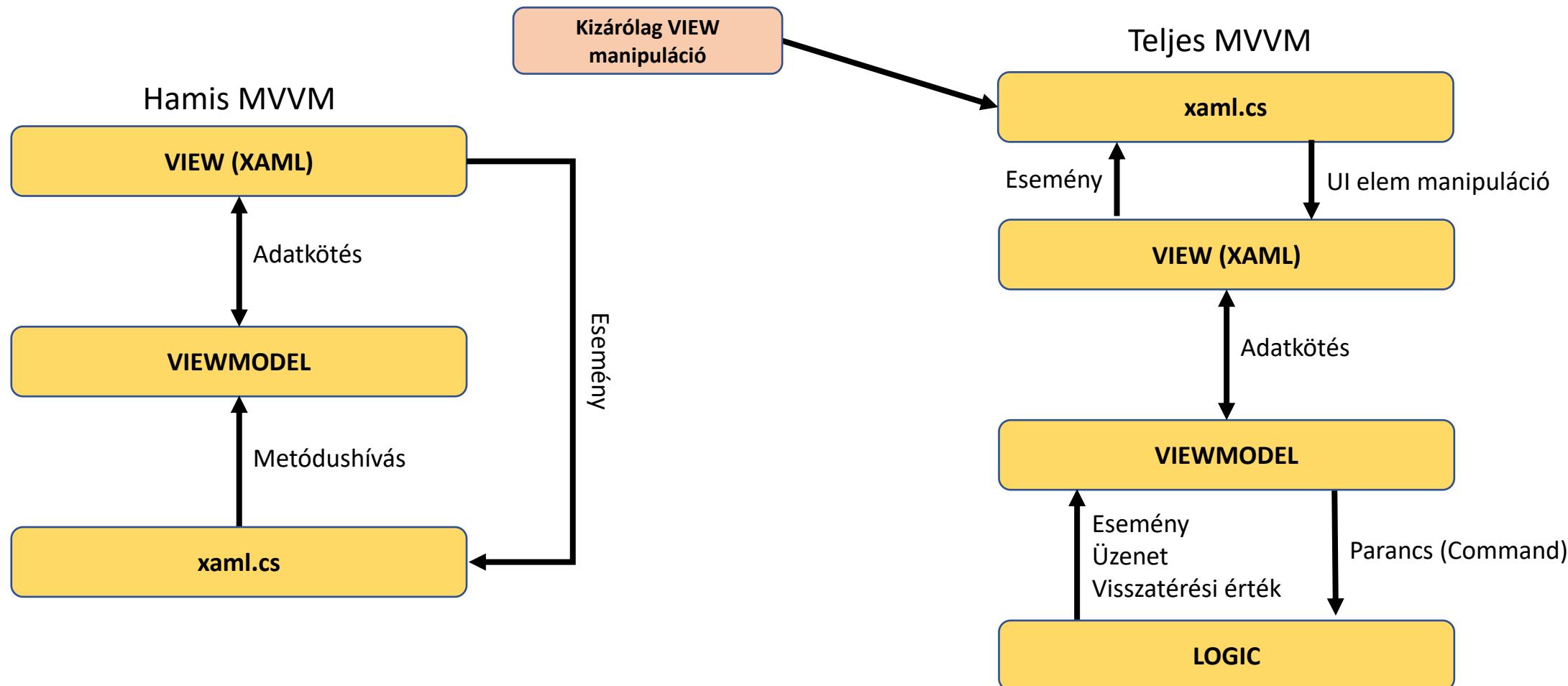


- **WPF ViewModel:** gyakorlatilag egy Logic, amit DataContext-nek használunk a Window szintjén

```
<Window.DataContext>
    <local:ViewModel />
</Window.DataContext>
```

- Gyűjtemények → **ObservableCollection<T>**
  - Entitások → **INotifyPropertyChanged** megvalósítás
  - Metódusok → Eseményekből hívva
  - Tulajdonságok → UI vezérlőkre kötve
- 
- **Ez még nem MVVM pattern**
    - Ez egyelőre a logika leválasztása
    - UI függ a logikától még → hiszen az eseménykezelőkben név szerint metódusokat hívunk



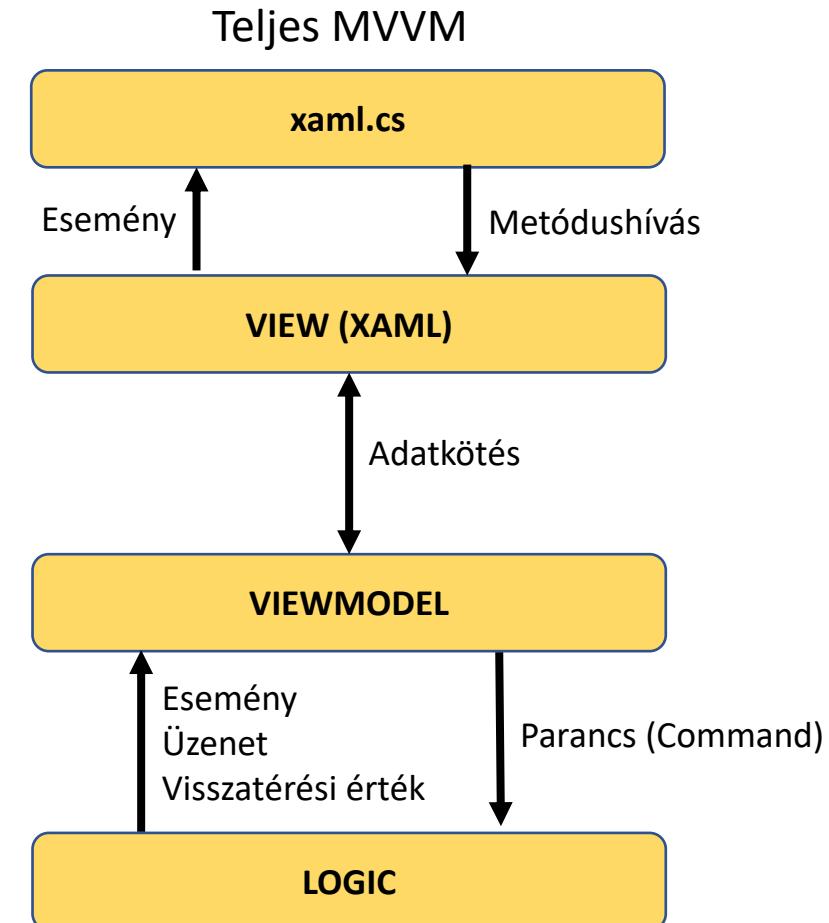


- Szabályok**

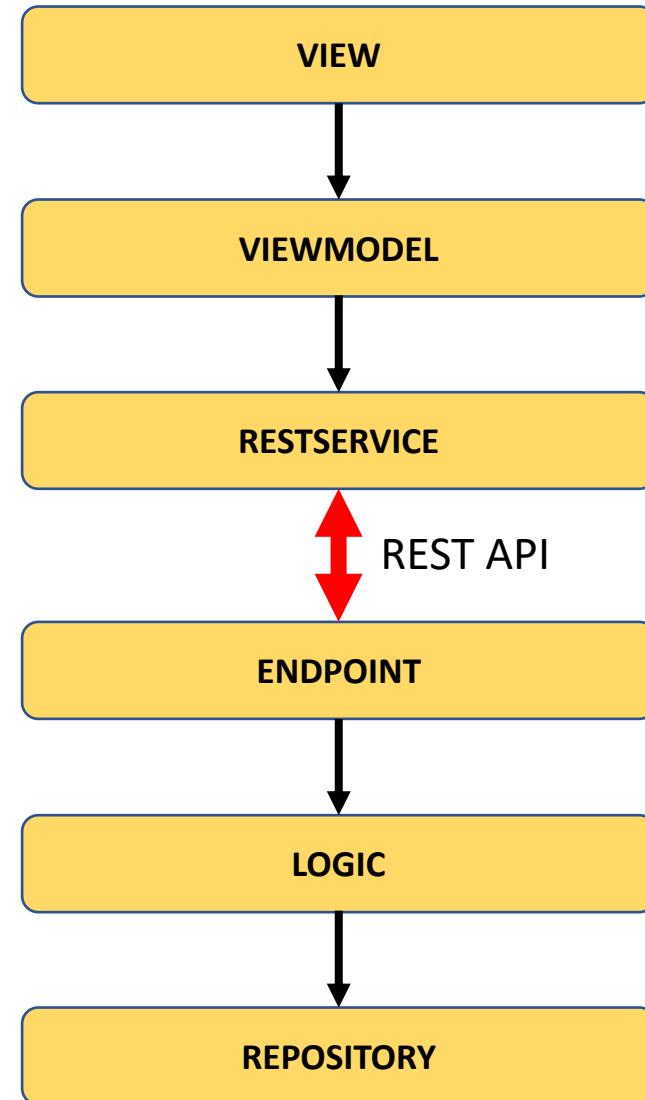
- **VIEW** csak a **VIEWMODEL-t** ismeri
- **VIEW** eseményei adatkötéssel vannak kötve a **VIEWMODEL** valamelyik **ICommand** típusú tulajdonságához
- **VIEWMODEL** tudja, hogy adott **Commandra** melyik **Logic** metódust kell hívni és milyen paraméterekkel
- **LOGIC** nem tud a **VIEWMODEL-ről**
- A **LOGIC** teljesen átvihető bármilyen nem GUI alkalmazásba is

- Xaml.cs**

- Csak VIEW segítése
- Alapvetően szinte üres



- A HFT tantárgyon fejlesztettünk egy üzleti logikát API végponttal
- **Cél**
  - Konzol kliens kiváltása WPF klienssel
- **Vastagkliens fejlesztés**
  - WPF app DLL-ként megkapja az alsóbb rétegeket
  - Egy eszközön lehet futtatni a teljes stacket
  - Túl sok értelme nincs
- **Vékonykliens fejlesztés**
  - WPF app API-kérésekkel éri el a Controller réteget
  - A Logic és az alatta lévő rétegek a szerveroldal
  - A WPF app a kliensoldal
  - Több WPF kliens is csatlakozhat egyidőben!



- **GOF: Gang of Four**
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software
- Gyakorlatilag minden tervezési mintákkal foglalkozó cikk, könyv és tanfolyam erre a műre hivatkozik
- De tudnunk kell: 1994-ben íródott, sok minden a modern programozási nyelvekre kell formálni belőle
- Jellemzőik
  - Újrafelhasználás maximális kiszolgálása
  - Örökösre vagy kompozícióra építenek

- **Elméletben**

- Újratervezés/módosítás/előretervezés során azonosítsuk a komponenseket, amelyeket javítani tudunk
- Válasszuk ki a megfelelő tervezési mintát és alkalmazzuk

- **Gyakorlatban**

- Fejlesztés közben nem figyelünk a baljós jelekre (code smells)
- Majd a dolgok „felrobbannak” és hónapokig/évekig szenvedünk a rosszul tervezett és megírt kóddal
- Refaktorálunk
- Mindig kicsit jobb lesz
- Majd olvasunk egy mintáról és rádöbbenünk, hogy eleve így kellett volna megírni és teljesen megoldaná az adott gondunkat

23 db minta		Cél		
		Létrehozási/Creational	Strukturális/Structural	Viselkedési/Behavioral
Hatókör	Osztály	Factory method	Adapter	Interpreter Template method
	Objektum	Abstract factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

- **Probléma:** Az objektumainkat gyakran bonyolult létrehozni és a konstruktur nem elég flexibilis ehhez
- Példa: egy pont a síkban
  - Létrehozható Descartes koordinátákkal (X és Y)
  - Létrehozható Polár koordinátákkal (szög és távolság)
  - Mind a kettő esetben 2 db double értéket várnánk
  - Nem hozhatunk létre két konstruktort ugyanolyan típusú paraméterekkel...
- **Megoldás:** egy külső **PointFactory** osztály, ami segít nekünk egy **Pointot** elkészíteni

# Abstract Factory

- **Probléma**

- Különböző feltételek alapján más és más objektumokat szeretnénk szolgáltatni
- Nem szeretnénk hatalmas if-ekkel dolgozni
- Pl: egy stringtől függ, hogy milyen osztály példányosítunk (Manager, Developer, Tester)
- Pl: egy attribútumtól függ, hogy milyen validációs osztályra van szükségünk

- **Megoldás**

- Egy ősfactory – sok leszármazott factory
- Dictionary vagy reflexió azonosítja a paraméter függvényében a megfelelő factory-t

- **Probléma**

- Egy objektum gyártásához rengeteg paraméter kell
- Ez gigászi konstruktorokat szül
- Nem is kell gyakran minden paraméter
- Opcionális paraméter hell kerülendő
- Null értékek kerülendőek

- **Megoldás**

- Üres konstruktor
- Láncolható gyártófüggvények, amelyek beállítják a paraméterek egy halmazát
- A gyártás kiszervezése egy különálló osztályba

- **Probléma**

- Jól jönne egy adott helyzetben statikus osztály használata, mert minden ugyanaz az állapottér kell
- Nem szeretnénk viszont statikus osztályt készíteni, mert nem tesztelhető
- Kellene egy olyan megoldás, hogy egy osztálytól minden ugyanazt a példányt kérhessük el
- Pl: adatbázis kapcsolattartó osztály (DbContext), konfigurációs beállítások, stb.

- **Megoldás**

- Osztály létrehozása
- Egy statikus adattag létrehozása neki
- Abban egy példány létrehozása
- Mindig ennek a példánynak a szolgáltatása

- **Probléma**

- Nem szeretnénk mindig objektumokat építeni a nulláról
- Jó lenne pl. egy régi objektumot lemásolni és csak az eltérő tulajdonságokat módosítani
- Pl: tantárgyi követelmények → lemásoljuk a tavalyit és néhány apróság változik (évszám, pontszámok)
- Ezt leprogramozni nem triviális, mert alapvetően shallow copy objektumokat kapunk
  - Jelentése: referencia típusoknál csak a referenciát kapjuk másolatként → ugyanarra az adatra mutat a két objektum
  - Egy objektum általában elégé összetett, van sok érték és sok referencia típus benne

- **Megoldás**

- Osztályok szintjén Deep Copy metódusok létrehozása
- Vagy serializáció, majd deserializáció

# Köszönöm a figyelmet!

Kérdés esetén e-mailben szívesen állok rendelkezésre.



# Szoftvertechnológia

# **MODUL 6**

**Strukturális tervezési minták**

**Adapter**

**Bridge**

**Composite**

**Flyweight**

# GOF minták

23 db minta		Cél		
		Létrehozási/Creational	Strukturális/Structural	Viselkedési/Behavioral
Hatókör	Osztály	Factory method	Adapter	Interpreter Template method
	Objektum	Abstract factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

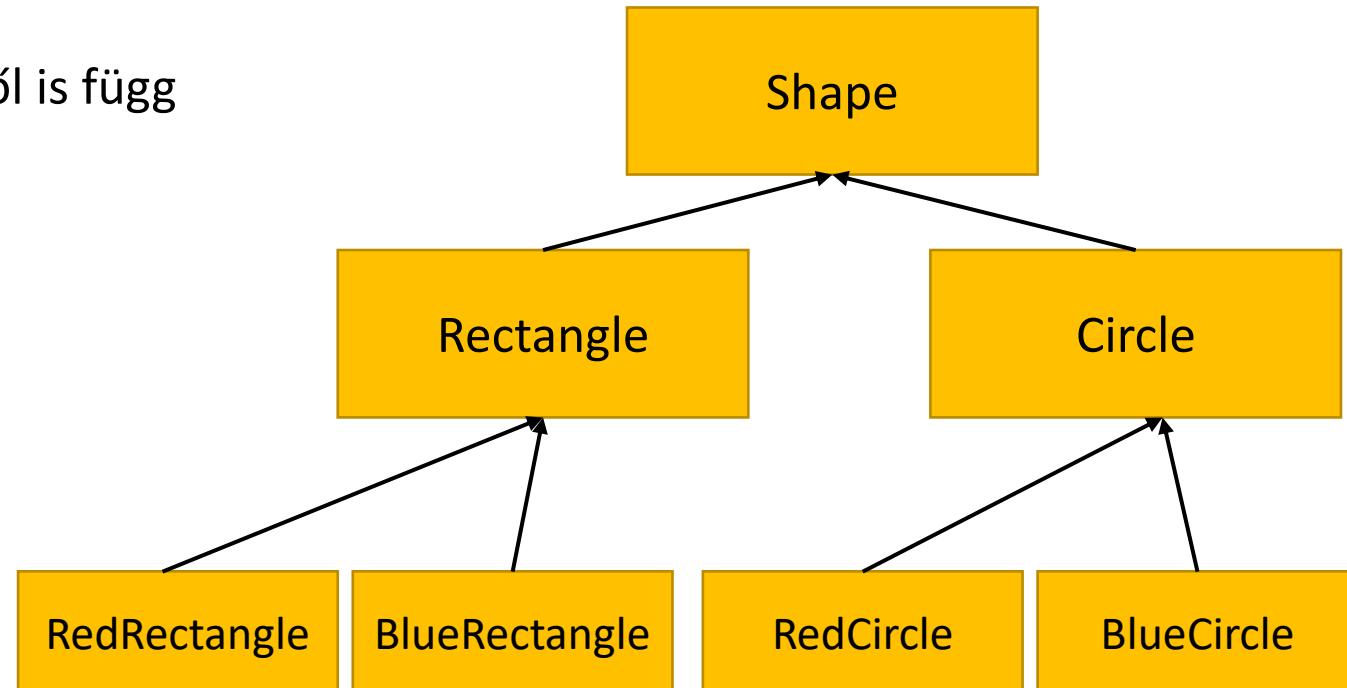
- **Probléma:** Össze szeretnénk kötni két rendszert, amelyek nem kompatibilisek
- **Példa**
  - Egy adatmegjelenítő osztály az adathalmazt **INameData** interfészen át várja
  - Egy adatforrás interfész az adathalmazt **INameSource** interfészen át biztosítja
  - Az adatmegjelenítőnek nem adható át az adatforrás
- **Megoldás**
  - Gyártunk egy **NameAdapter** osztályt
  - Ez megvalósítja az **INameData** interfészt
  - Forrásként várja az **INameSource** interfészt
  - Megvalósítja a konvertáló logikát
- Való élet példa: adapter kábelek

- **Probléma**

- Észrevesszük, hogy egy osztály két jellemzőtől is függ
- Pl: alakzatok → szín és forma
- Valami ilyen hierarchiát építettünk

- **Megoldás**

- Szét kell bontani az osztályt
- A forma osztály várja interfészen keresztül a szín osztályt
- Kompozícióval lehessen összeépíteni őket



- **Probléma**

- Nehezen tudunk az objektumainkból hierarchikus rendszert építeni
- Pl: részlegek és dolgozók korrekt ábrázolása
- Egy részfa vagy akár egy levélelem is ugyanazt a szolgáltatáskészletet nyújtsa

- **Megoldás**

- Fa szerkezet építése
- Egy csomópontnak tetszőleges mennyiségű gyermekelme legyen
- A csomópontok és levél elemek is ugyanazt az interfészt valósíták meg
- Lehessen rekurzívan bejárni

- **Probléma**

- Szeretnénk optimalizálni az alkalmazást, hogy kevesebb memóriát használjon
- Ezt a CPU kárára tudjuk megtenni
- Nincs egy konkrét megoldás – sok trükköt biztosít a **FlyWeight** minta

- **Megoldás(ok)**

- On-the fly property-k
- Objektumok közös részeinek eltárolása egyszer
- Újrahasznosított objektumok

# Köszönöm a figyelmet!

Kérdés esetén e-mailben szívesen állok rendelkezésre.



# Szoftvertechnológia



ÓBUDAI EGYETEM  
NEUMANN JÁNOS INFORMATIKAI KAR

# MODUL 7

**Strukturális tervezési minták**

**Facade**

**Proxy**

**Decorator**

# GOF minták

23 db minta		Cél		
		Létrehozási/Creational	Strukturális/Structural	Viselkedési/Behavioral
Hatókör	Osztály	Factory method	Adapter	Interpreter Template method
	Objektum	Abstract factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

- **Probléma:** Van egy probléma, amit mindig nehézkes megoldani, mert több osztály/alrendszer is kell hozzá, amelyeket körülözényes konfigurálhatni
- **Példa**
  - Egy webes végpont JSON formában adatokat biztosít
  - Szükséges hozzá egy  **WebClient**, hogy letölthessük a JSON szöveget
  - Szükséges hozzá egy  **JsonConverter**, hogy deserializálhassuk
  - Ezeknek a használatát minden lepuskázzuk egy korábbi projektünkben ☺
- **Megoldás**
  - Gyártunk egy  **SimpleJsonConsumer<T>** osztályt
  - Legyen képes  **nagyon könnyen** visszaadni a  **List<T>**-t egy URL cím alapján
  - Megjegyzés: extra logic kódot  **nem adunk hozzá**, csak egyszerűsítünk!

- **Probléma**

- Adott egy rendszer, amelyet módosítani nem tudunk (pl: egy CRUD subsystem)
- Interfészen keresztül érjük el
- Az alkalmazásunk már üzemel
- De szeretnénk beinjectálni **többlet kódot**, ami a metódushívások előtt lefut (pl. jogosultságkezelés, telemetria, stb.)

- **Megoldás**

- Implementálunk egy új osztályt az eléri kívánt rendszer interfésze alapján
- Az eléri kívánt rendszer ebbe kerül be kompozícióval
- A rendszerrel azonos metódusaink lesznek, amelyek meghívják a rendszer ugyanilyen nevű metódusait
- De a meghívás előtt futtathatunk többlet kódokat
  - Pl: jogosultságkezelés → protection proxy
  - Pl: a helyi metódus egy távoli metódust hív API-n keresztül → remote proxy

- **Probléma**

- Szeretnénk egy objektumot ellátni többlet jellemzőkkel
- Pl: egy képet el szeretnénk látni kerettel → ő maga is legyen egy kép!
- Pl: egy képet el szeretnénk látni felirattal → ő maga is legyen egy kép!
- Pl: egy hallgatót el szeretnénk látni Erasmus adatokkal → és utána ugyanolyan hallgatóként tudjuk használni

- **Megoldás**

- Leszármazott osztály készítése vagy közös interfész megvalósítása
- Kompozícióval a kibővítendő objektum eltárolása
- Többletinfók hozzáadása
- Új objektum használata gond nélkül

# Köszönöm a figyelmet!

Kérdés esetén e-mailben szívesen állok rendelkezésre.



# Szoftvertechnológia

# MODUL 8

**Viselkedési tervezési minták**

**Iterator**

**Chain of responsibility**

**Visitor**

**Command**

**Observer**

**Mediator**

# GOF minták

23 db minta		Cél		
		Létrehozási/Creational	Strukturális/Structural	Viselkedési/Behavioral
Hatókör	Osztály	Factory method	Adapter	Interpreter Template method
	Objektum	Abstract factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

- **Probléma:** Legyen bármilyen gyűjteményünk (lista, fa, gráf, hasítótábla, stb.) ezeket szeretnénk egy bejárható interfészen keresztül elérni
- **Megjegyzés**
  - Korábban már tanultunk róla a láncolt listánál
  - LINQ is erre épít
  - **IEnumerable<T>** interfészen át érjük el a gyűjteményeinket
- **Megoldás**
  - Az adatszerkezetünkön implementáljuk az **IEnumerable<T>** és egy külső bejáró osztályon az **IEnumerator<T>** interfészt
  - Az **IEnumerator<T>** előírja az alábbi metódusokat:
    - **void Reset()** → gyűjtemény elejére visszaállás
    - **bool MoveNext()** → következő elemre lépés
    - **T Current** → visszaadja az aktuális elemet
  - Vagy **yield return** → lásd példa

# Chain of responsibility I.

- **Probléma**

- Egy olyan folyamatot implementálunk, ahol egy objektum sok részfeladaton megy keresztül
  - Pl: 10 db validációt egymás után meghívunk rajta → ha bármelyik hibára fut, nem nézzük a többet (feltételes továbbpasszolás)
  - Pl: 10 db naplózó metódusnak egymás után átadjuk az objektumot → mindenépp mindegyiken végigmegy (feltétel nélküli továbbpasszolás)
- Első esetben egy nagyon összetett elágazás jön létre
- Utóbbi esetben pedig 10 db metódushívás egymás alatt

- **Megoldás**

- Készítünk a validáló/logoló osztályokból egy gyűjteményt
- Végigmegyünk az elemein és mindegyiknek átadjuk az objektumot

- **Probléma**

- Egy olyan folyamatot implementálunk, ahol egy fizetési objektum egy döntési hierarchián fut végig
  - Osztályvezető → 10.000 ft alatt hagyhatja jóvá
  - Középvezető → 100.000 ft alatt hagyhatja jóvá
  - Felsővezető → minden más esetben hagyhatja jóvá
- Csak a közvetlen feletteshez intézünk kérelmet, láthatatlan számunkra, hogy ő maga saját hatáskörben jóváhagyta, vagy pedig a saját feletteséhez fordult → csak kapunk egy választ, hogy igen/nem

- **Megoldás**

- Elkészítünk egy Boss osztályt
- A példányok tárolják a saját pénzügyi hatókörüket
- A példányok tárolják a saját felettesüket
- A példányok döntést hoznak vagy továbbítják a kérelmet felfele

- **Probléma**

- Szeretnénk a **hívót** és a **hívottat** szétválasztani
- A **hívó (Logic)** tudhat a **hívottról (Subject)**, de fordítva tilos
- A hívott dönthessen róla, hogy vele éppen lehet-e dolgozni
- Működjön az egész gyűjteményekkel is (több **hívó** és több **hívott**)

- **Megoldás**

- Interfészeken át érjék el egymást
- **Hívottnak** legyen: **Accept()** metódusa
- **Hívónak** legyen: **Visit()** metódusa
- A **hívott** az **Accept()** metódusában döntést hoz és egyben meghívja a **hívó Visit()** metódusát

- **Probléma**

- A Subject állapotváltozásairól szeretnénk különböző feliratkozókat értesíteni
- De nem igazán kéne tudnunk róla, hogy milyen feliratkozók vannak és hogy vannak-e
- Pl: változott egy személy fizetése → minden felületen, adattárolóban módosítsuk

- **Megoldás**

- Feliratkozó osztályok valósítsanak meg egy **ISubscriber** interfészt
- Ez írjon elő egy **StateChange()** metódust
- A **subject** kezelje a feliratkozókat (**Subscribe()**, **UnSubscribe()**)
- Állapotváltozáskor hívja meg az összes feliratkozó **StateChange()** metódusát
- A feliratkozók tegyék meg a frissítési lépésekét

- **Probléma**

- Egyirányú függés van két nagy réteg között
- **UI** ismeri a **Logic**-ot (példában **Repository**-t)
- Ha lecseréljük egy másik **Repository**-ra, akkor nagyon sok helyen a **UI** kódját át kell írnunk

- **Megoldás**

- Közvetve ismerje csak a **UI** a **Repository**-t
- A közvetítő osztályban kelljen átírni bármit, ha módosul a **Repository**
- A közvetítő osztályba pedig extra kód is legyen írható

- **Probléma**

- Egyirányú függés van két nagy réteg között
- Ne legyen semmilyen irányú függés
- Egy közvetítő osztályon keresztül lehessen csak beszélgetni két osztálynak

- **Megoldás**

- Legyen egy **Mediator** osztályunk
- Lehessen regisztrálni egy üzenetfolyamára
- Lehessen beküldeni egy üzenetet valamelyik üzenetfolyamára

# Köszönöm a figyelmet!

Kérdés esetén e-mailben szívesen állok rendelkezésre.



# Szoftvertechnológia

# MODUL 9

Viselkedési tervezési minták

**Strategy**

**Template method**

**Memento**

**State**

**Interpreter**

# GOF minták

23 db minta		Cél		
		Létrehozási/Creational	Strukturális/Structural	Viselkedési/Behavioral
Hatókör	Osztály	Factory method	Adapter	Interpreter Template method
	Objektum	Abstract factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

- **Probléma**

- Szeretnénk különböző titkosításokat használni (pl. cézár, pa-le-ri-no, stb.)
- Szeretnénk a logikában titkosítva levelet küldeni
- Szeretnénk a programunkhoz a jövőben új titkosítási módszereket adni flexibilisen

- **Megoldás**

- Ős referenciát használunk mindenhol
- Leszármazottakkal felül lehessen definiálni az absztrakt titkosítás metódusát az ősnek

# Template method

- **Probléma**

- Szeretnénk bizonyos folyamatok egyes részeit a későbbiekben módosítani
- Pl: egy **PersonLogic** képes szűréseket végezni embereken
  - Leszármazottakkal lehessen felüldefiniálni a beolvasást
  - Leszármazottakkal lehessen felüldefiniálni a validálást

- **Megoldás**

- A logika meghívja a saját virtuális **Import()** és **Validate()** metódusait
- Ezeket leszármazotttal felül lehet írni, ha szeretnénk

- **Probléma**

- Undo funkciót szeretnénk implementálni egy entitásra
- Kívülről ne is nagyon lássuk, hogyan oldja meg
- Legyen képes bármikor visszaállni az előző állapotába
  - Kiegészítés: akár legyen képes BÁRMELY előző állapotra visszaállni

- **Megoldás**

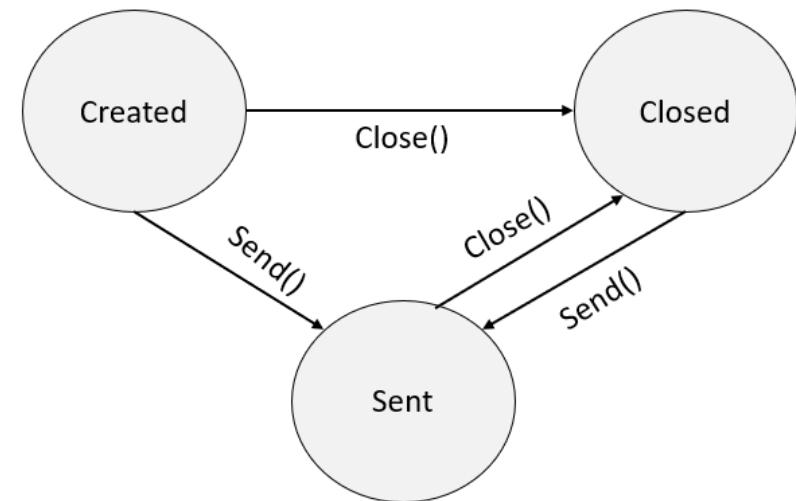
- Elkészítünk egy **Memento<T>** osztályt
- Az entitás kompozícióval eltárolja a **Memento<T>** osztályt (ami egy „state store”)
- Felhasználjuk az entitásban a prototype mintát
- A deep copy-t elmentjük a **Memento**-ba
- **Memento** biztosítson **Undo()** metódust

- **Probléma**

- Különböző állapotváltozásokat szeretnénk egy objektumban tárolni
- Pl: hibajegy
  - Inicializálás → Created state
  - Send() → Send state
  - Close() → Closed state
- State-machine diagrammal tervezük meg
- Lehet-e pl. Closed-ból Send()-elni és ezáltal újranyitni?

- **Megoldás**

- Osztályba **ITicketState** interfészen át bevesszük az állapotot
- Létrehozzuk a **Send()** és **Close()** metódusokat
- Az **ITicketState**-ben is van **Send()** és **Close()** metódus
- minden állapotnak egy-egy implementációt készítünk



- **Probléma**

- Tetszőleges bemenetből tetszőleges kimenetet szeretnénk gyártani
- Pl: egy  $(3 + 4) - (2 + 2)$  stringből egy intet, aminek az értéke: 3
- Értelmező programok írásának OOP reprezentációja az **Interpreter** minta

- **Megoldás**

- Elkészítjük az írásjeleket reprezentáló osztályokat (**Token**)
- Elkészítünk egy **Lexer**-t
- Elkészítünk egy **Parser**-t

# Köszönöm a figyelmet!

Kérdés esetén e-mailben szívesen állok rendelkezésre.



# Szoftvertechnológia

# MODUL 10

**Domain-driven design**

**CQRS**

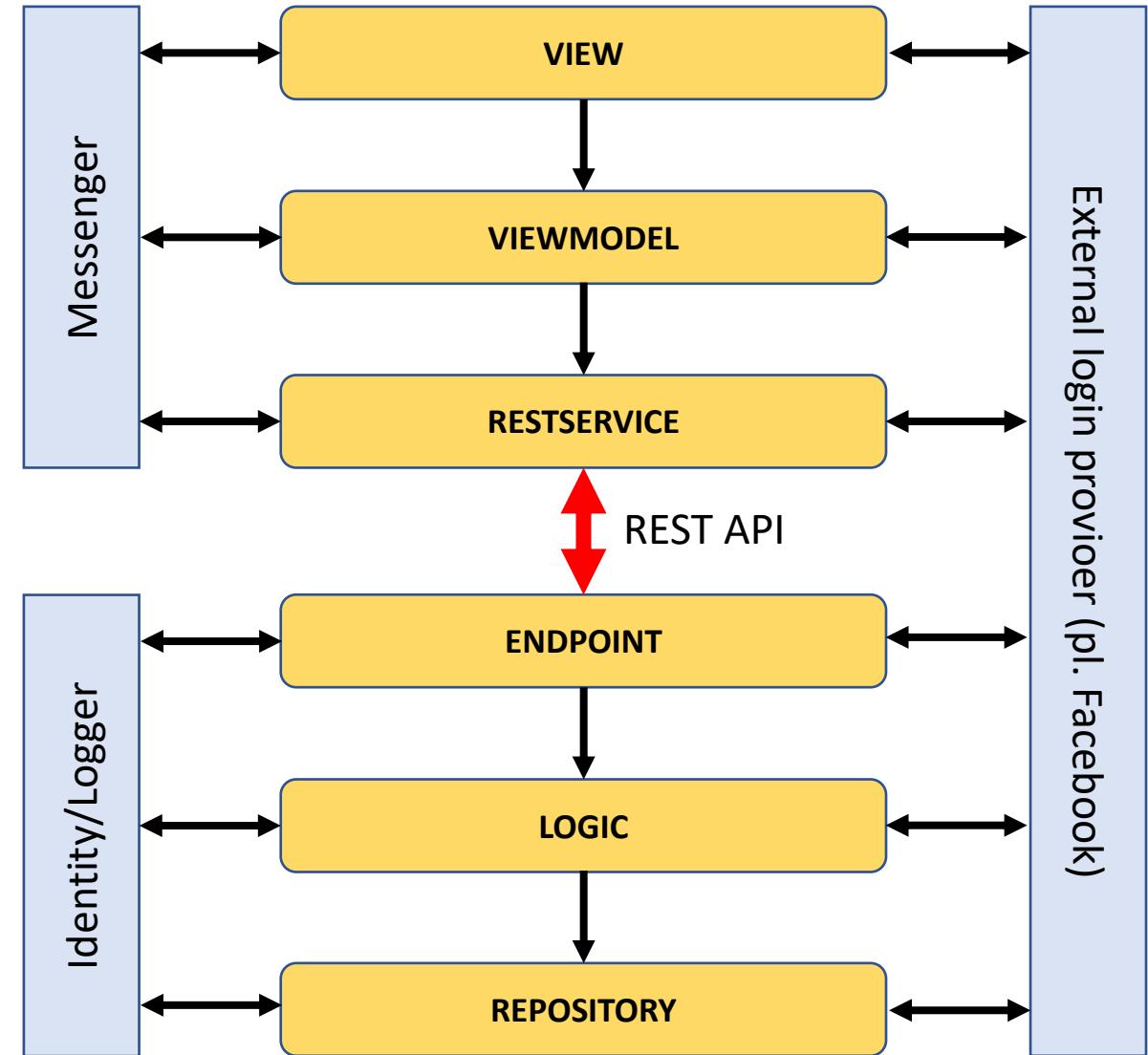
**Mikroservices**

**Event brokers**

- Adatelérés → Üzleti logika → Megjelenítés
- Ezek akkor igazán hatékonyak, hogyha
  - Felhasználó használja a rendszert
  - Alapvetően CRUD funkcionálisra van az egész kihegyezve
- Mi használhatja még a rendszert?
  - Automata tesztek
  - API végpontok
  - Scriptek
  - Belső és külső automatizmusok
- Vannak olyan komponensek, amelyekre minden rétegben szükség van
  - Naplázás
  - Security
  - Messenger

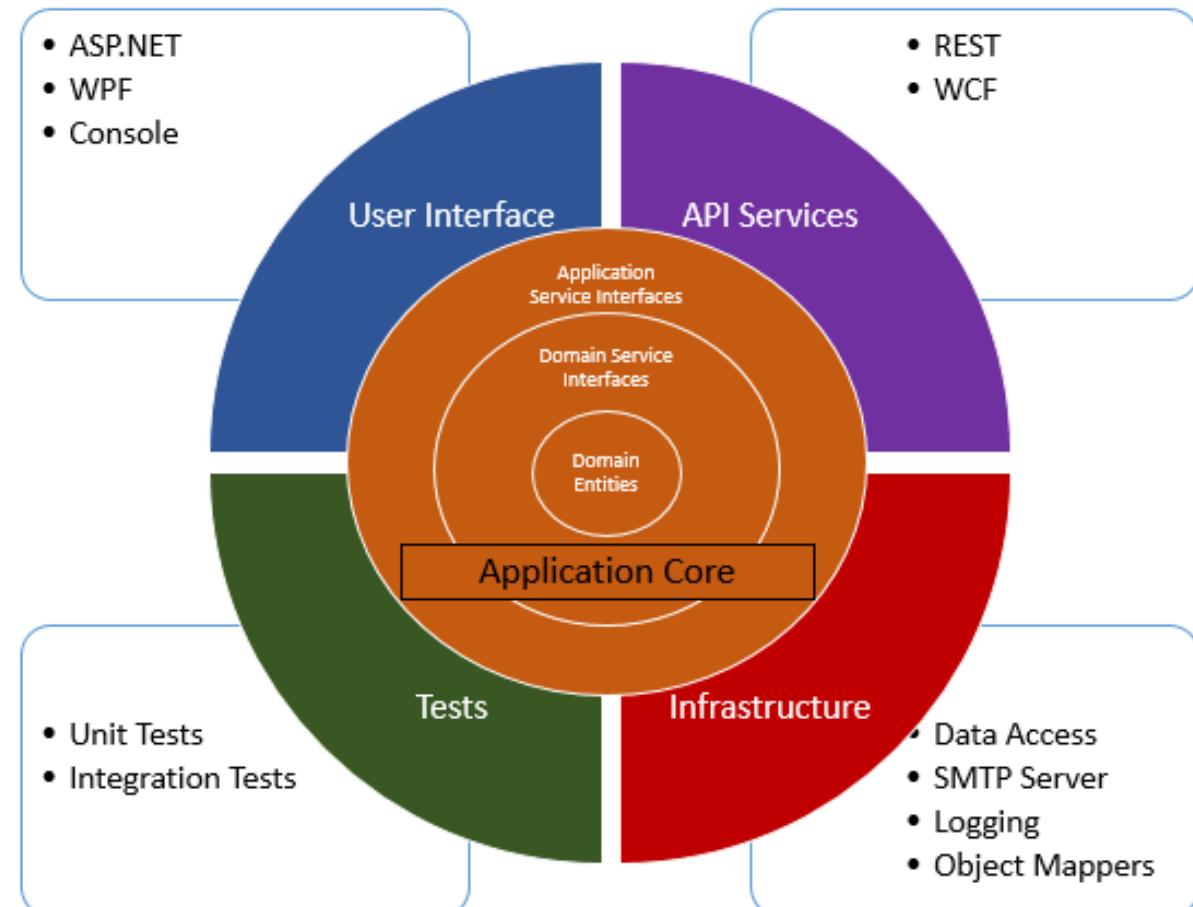
# Aspect-ek

- Olyan komponensek, amelyeket minden réteg használ
- Tipikus elvárások az aspect-ekkel szemben
  - Ne kövessenek el rétegsértést
  - Legyenek szűk funkcionálitásúak



# Onion architecture

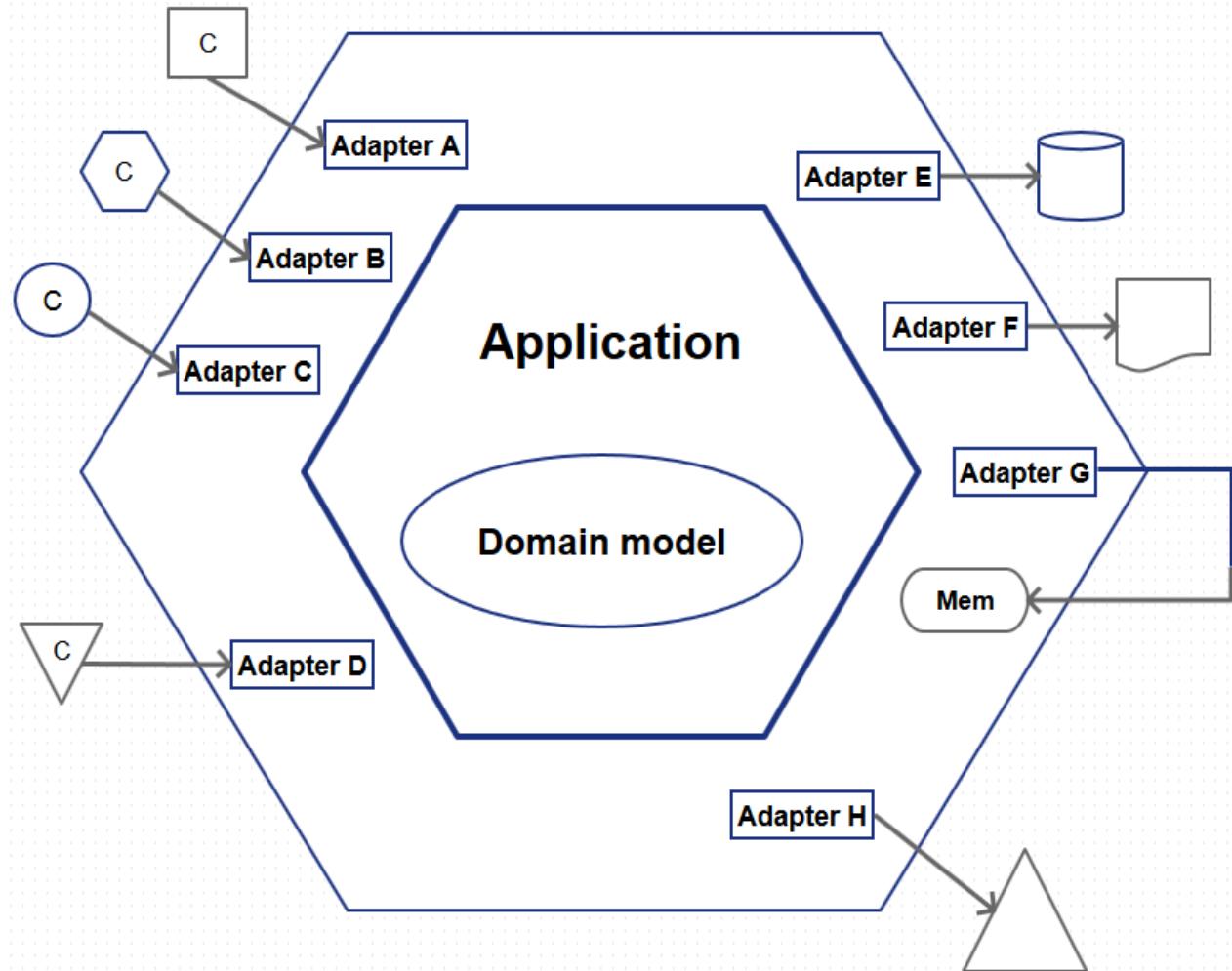
- Application core = Logic
- Logic felett több réteg is elhelyezkedik
- Mindegyik más-más célt szolgál ki
- Pl: webalkalmazás
  - Logic adott
  - Logic felett
    - MVC-vel webes UI
    - API végpontok mobilapphoz



# Hexagon architecture

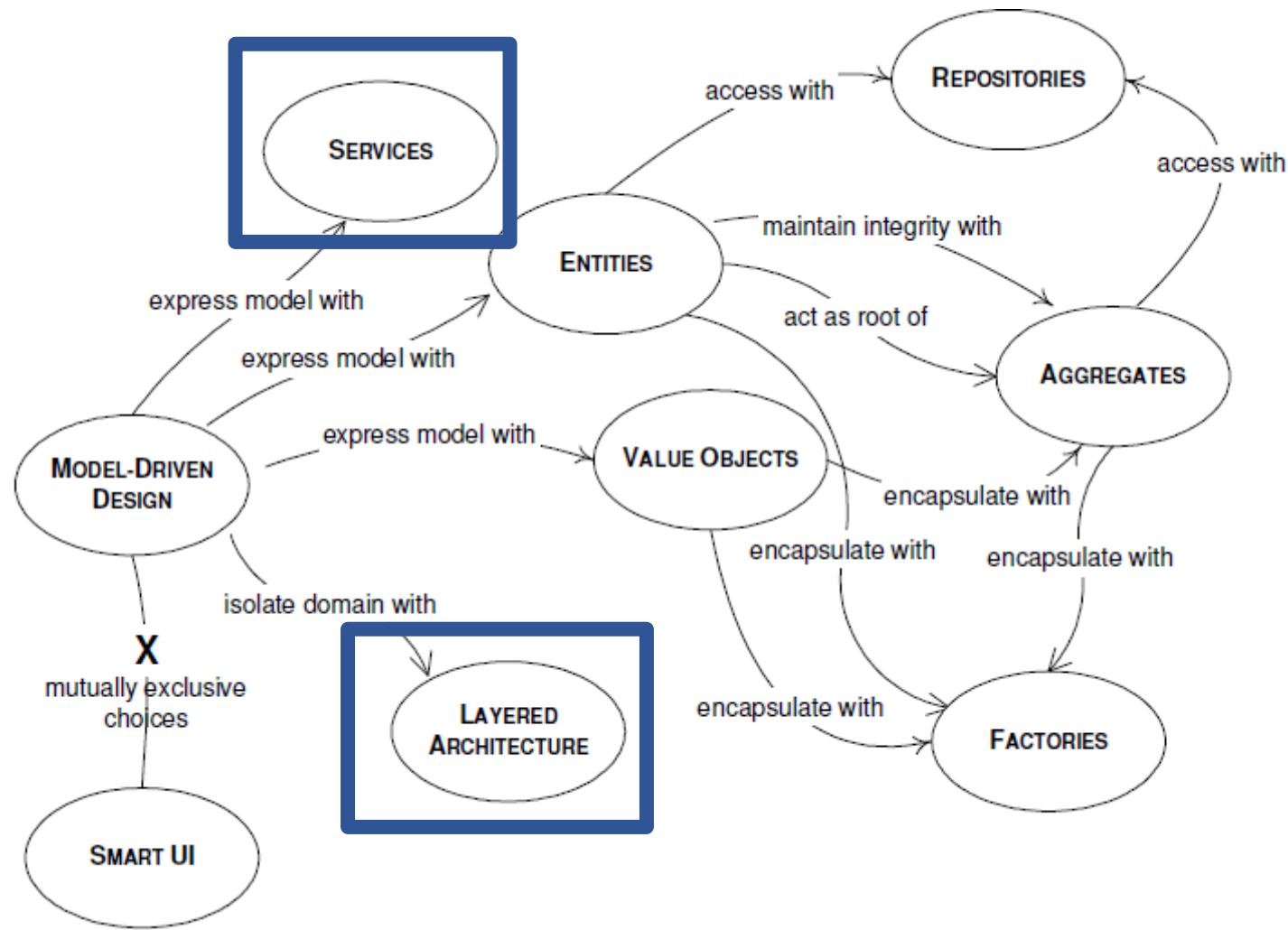
6

- Application core = Logic
- Logichoz külső rendszereket csatolunk adaptereken keresztül
- Külső rendszerek
  - Adattárolás
  - API végpontok
  - Webes UI
  - Email küldés
  - Logolás
  - Felhasználókezelés
  - Stb.



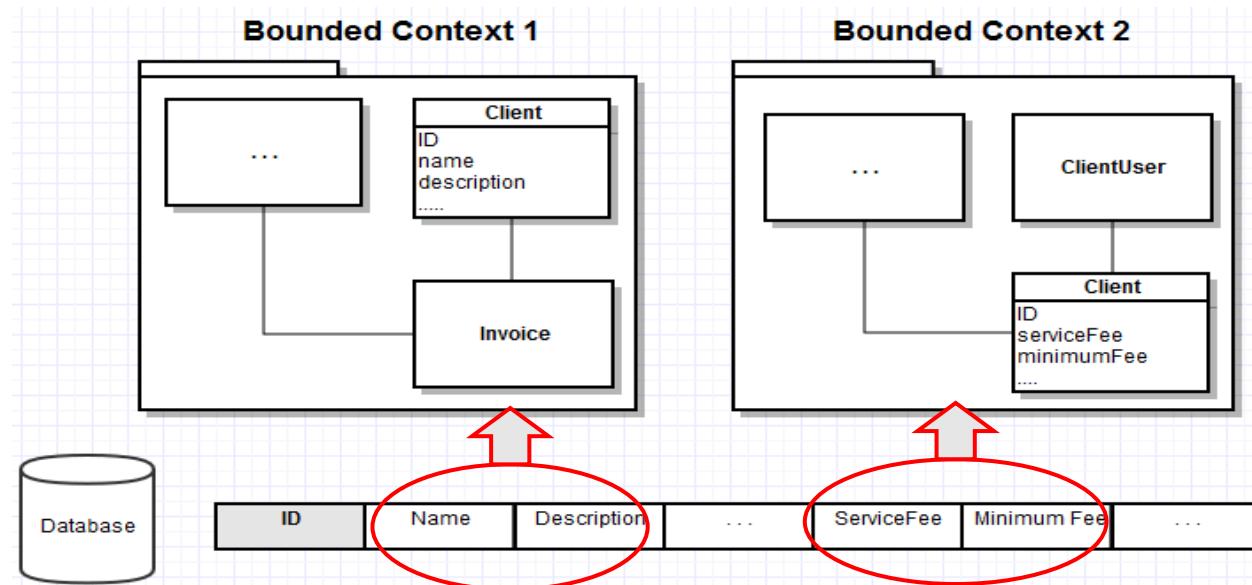
- Martin Fowler, Vaughn Vernon, Eric Evans, Udi Dahan → DDD
- Lényege: a rétegzés ne attól függjön, hogy MVC vagy API vagy bármilyen UI elérési technika
  - **Attól függjön a rétegzés, hogy mit akarok csinálni az adattal**
- Irodalom a téma körül:
  - Refactoring (Improving the Design of Existing Code, 2018 2nd ed)
  - <https://martinfowler.com/books/eaa.html>
  - Patterns of Enterprise Application Architecture (2003/2004) → újabb 51 db pattern!
    - Domain Logic Patterns
    - Data Source & Object-Relational Behavioral & Structural Patterns → ORM
    - Object-Relational Metadata Mapping Patterns
    - Web Presentation Patterns
    - Distribution Patterns
    - Offline Concurrency Patterns
    - Session State Patterns
    - Base Patterns

# Domain-driven design



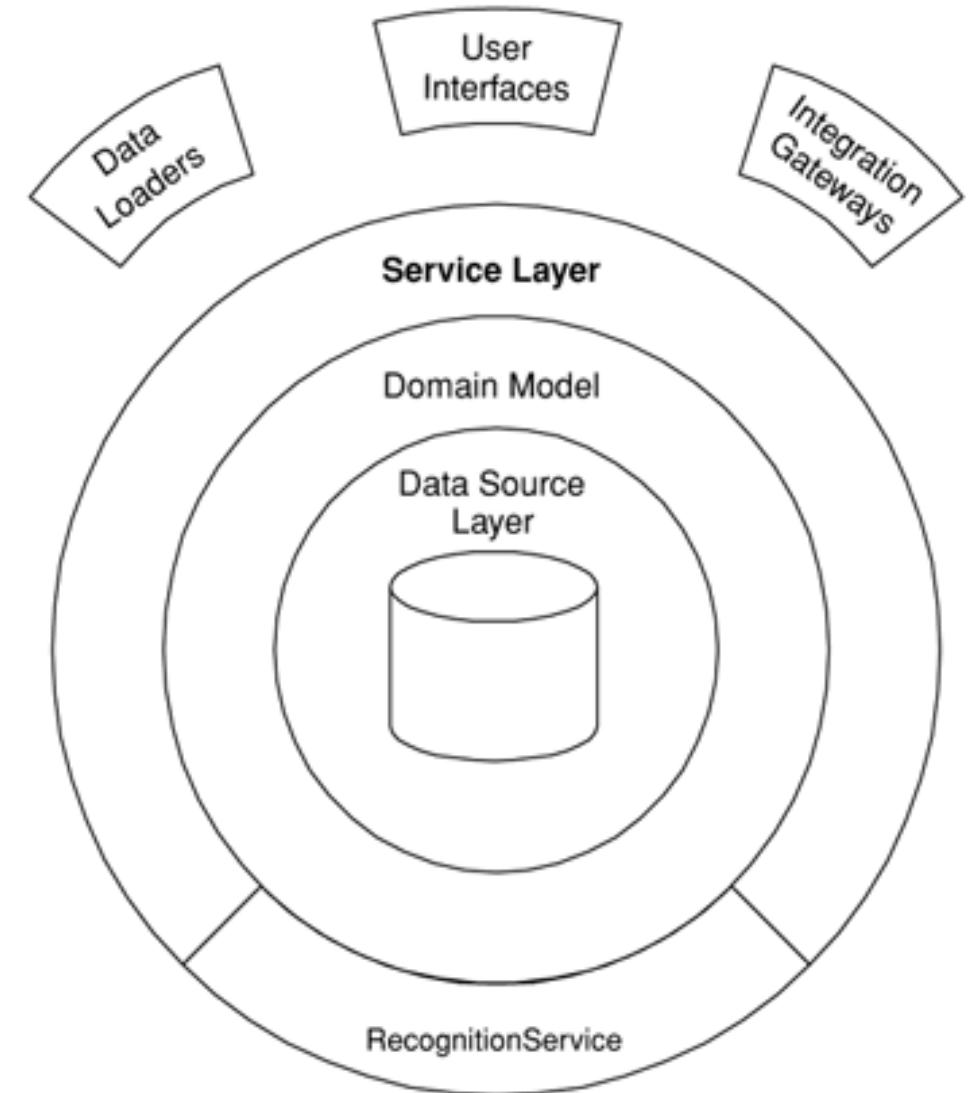
# Domain-driven design

- Nem az adatbázissal kezdjük a modellezést, hanem a funkciókkal
- **Bounded Context**-eket hozunk létre → Domain model lesz belőlük
  - Jelentése: Egy User tábla szerepelhet a Szállítás domain modelben és a Számlázás domain modelben is
  - Ez így DRY elveknek ellentmond...
  - De csak látszólag, mert a Data Mapper / ORM majd valójában ugyanarra az 1 db táblára mappeli le
  - Hibalehetőségek: Bloated domain objects (túl sok felelősség), Anemic domain objects (túl kevés felelősség)

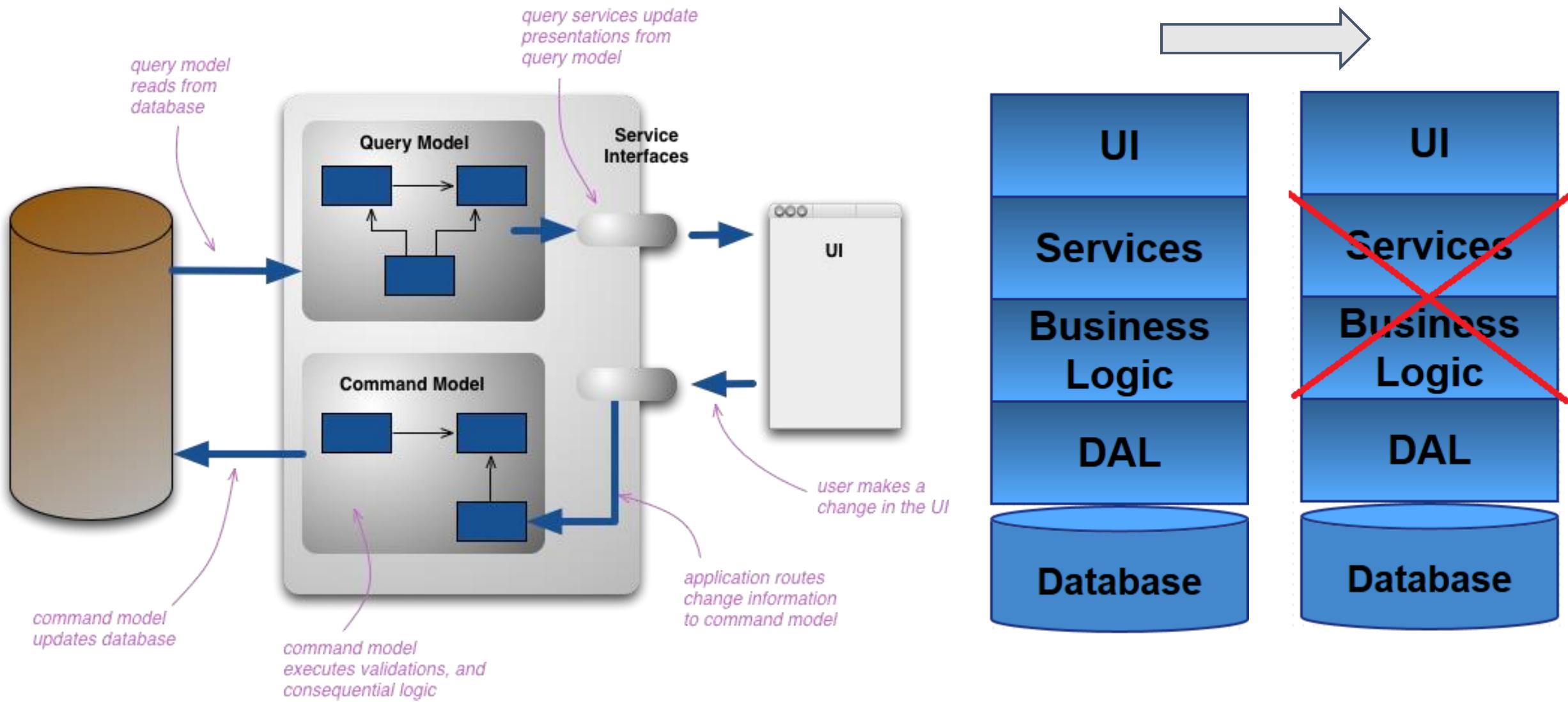


# Service Layer

- Domain model egységbezárja a Domain Entityket és azok üzleti műveleteit
- SOA filozófiát valósít meg (Service-Oriented Architecture) → Microservices
- Feladata: hívások fogadása, továbbítása a Domain Logic felé
- Feladata lehet még a tranzakciókezelés és a lock is
- Alsóbb rétegekben megjelenik ettől függetlenül az adatbázis szintű tranzakciókezelés is
- UI csak egy service a sokból



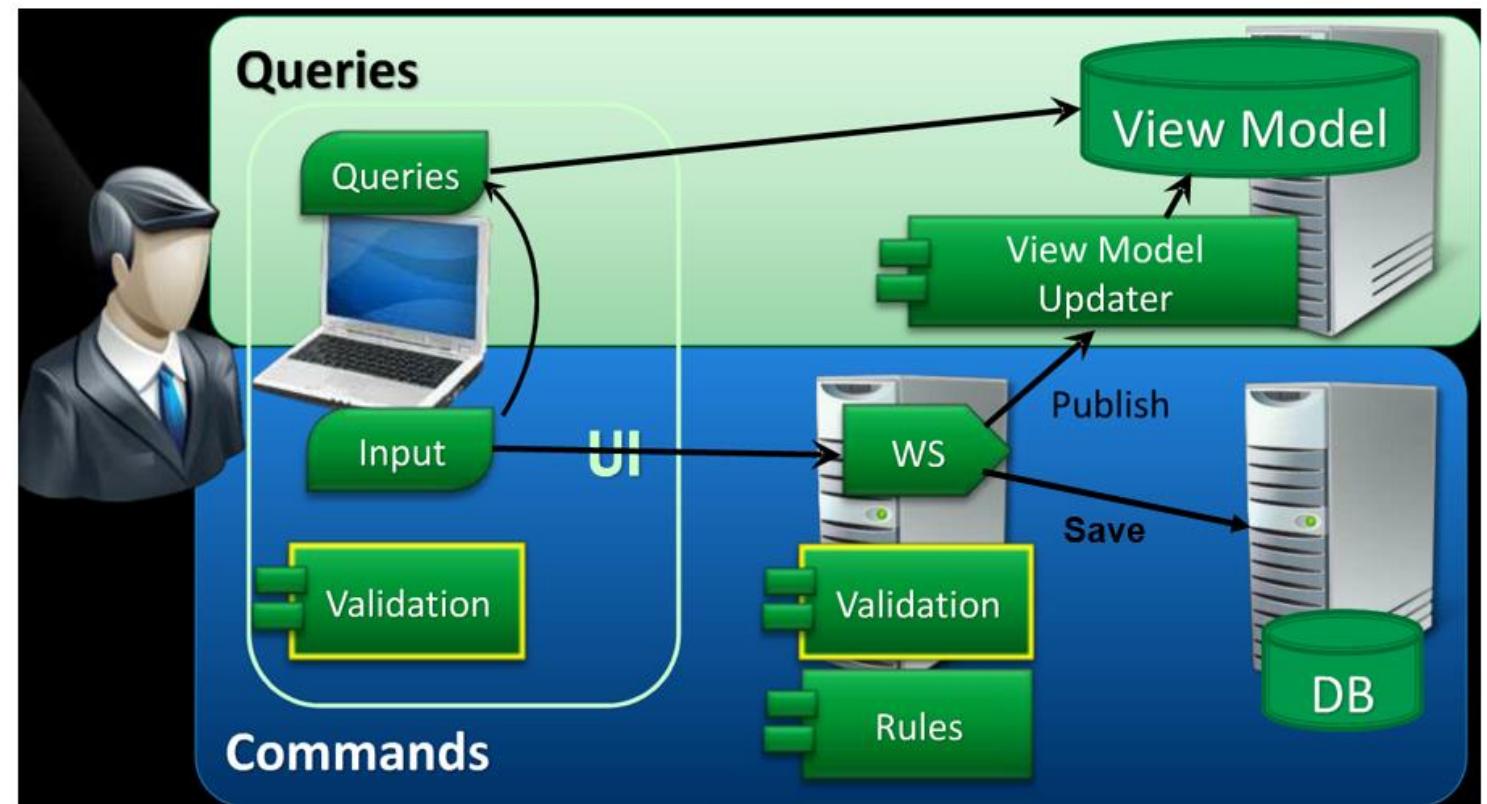
- Nagy rendszerek esetén általában SOK olvasási művelet és KEVÉS írási művelet történik
- Írási műveletek
  - Tipikusan egy bounded context-be akarunk írni (pl. egy számla létrehozása)
  - Kell minden alrendszer hozzá (jogosultság kezelés, validáció, tranzakciókezelés, stb.)
- Olvasási műveletek
  - „Dashboard” és „Reports” funkциonalitás nagyon gyakori
  - Általában több bounded context-ből kell összeszedni az adatokat (group by/join lassú...)
  - Egy csomó alrendszer kikerülhető akár (nem kell validáció, naplázás, tranzakciókezelés)
- Ez szétválasztható lenne két nagy alrendszerre
  - CQRS (Command-Query Responsibility Segregation)



# Olvasás gyorsítása

13

- A User lásson minden régi adatot
- Pl. Neptun → Képzettség adatok (dashboard) → sok bounded contextból áll össze
- Megjelenítése lehet, hogy 3-4 mp is lenne
- **Megoldás**
  - Persistent View Model → minden éjfélkor pl. egy adatbázisba mentsük ki az összes user dashboardját
  - Innen a betöltést 1 db select 1 db where záradékkal



- **Query oldal** – Olvasási műveletek

- Egyedi keresések problémája
  - Generálható minden éjjel a top 100 népszerű keresés találati oldala (pl: budai penthouse lakások)
  - A ritka keresések majd tovább tartanak (gond?)
  - Adatbázisok optimalizálhatóak keresésekre
  - Gyors keresésre optimalizált DB: ElasticSearch

- **Command oldal** – Írási műveletek

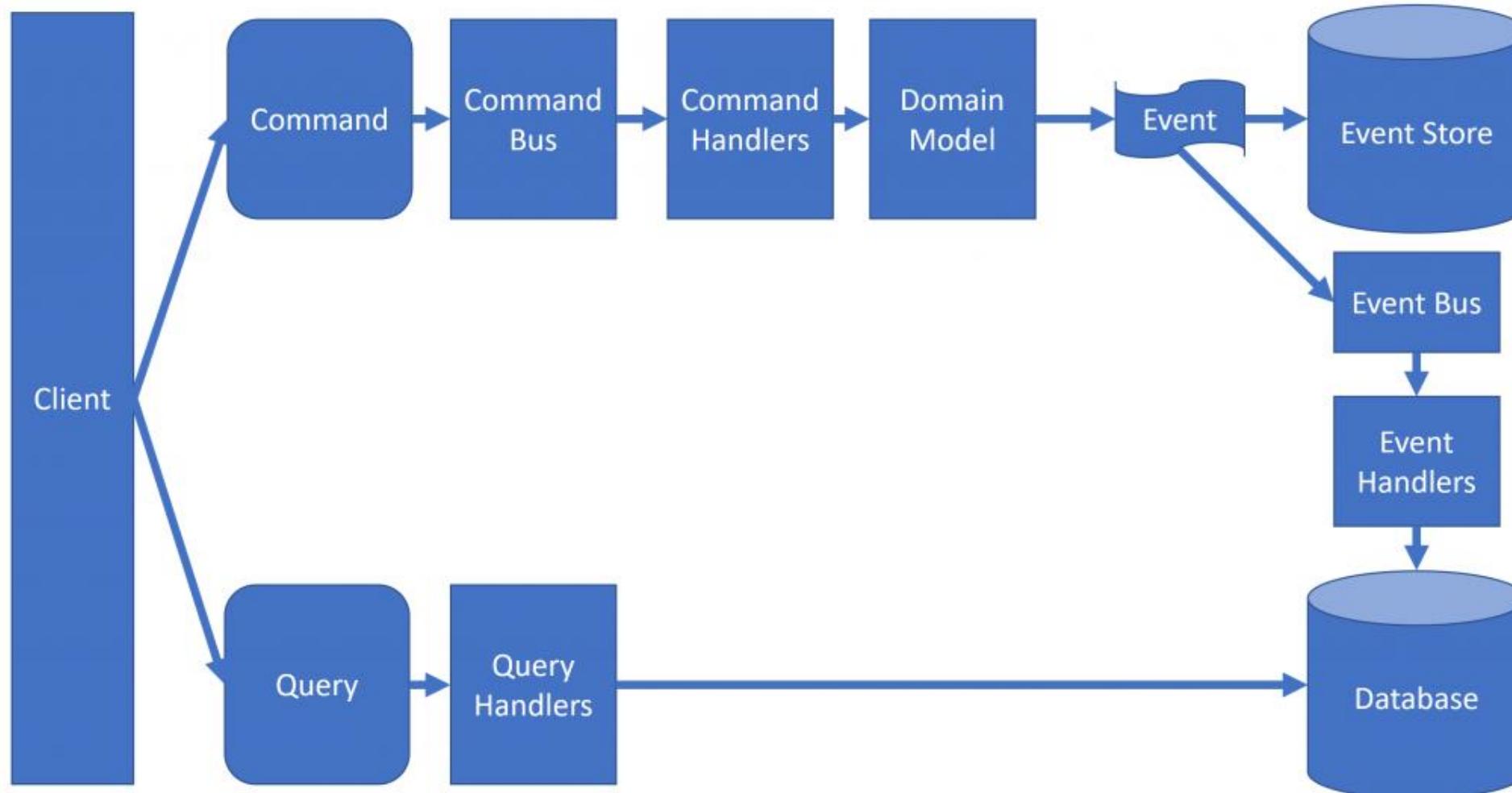
- Hibára futás ritka
- Szinkron hibajelzés feleslegesen lassít
- Aszinkron hibajelzés
  - Azt mondjuk, hogy sikeres a foglalás 😊
  - Ha netán valami baj van, akkor 5-10 perc múlva küldünk egy mailt, hogy mégis hiba történt
  - Aszinkron reagáló mechanizmusok → akár kézi megoldások
- Azért ez nem mindenhol használható!!! → jegyfoglalás, tárgyfelvételnél tipikusan nem
- Ahol szóba jöhet: hozzászólás, üzenetküldés, értékelés, megrendelés, jelentkezés, regisztráció, stb.

- Tipikus probléma, hogy gyorsabban jön az input, minthogy fel tudnánk dolgozni
- Pl: Egy szenzor akarna 1mp-enként adatot küldeni, de a szerver annyira túlterhelt, hogy 3 mp mülva jön meg a HTTP response
  - Gyakorlatilag feltorlódnak a kérések és használhatatlan lesz a rendszer
- Pl: Kijön az új Iphone, van belőle 100.000 db készleten
  - Éjfélről lehet előrendelni
  - Korábbi évek statisztikái alapján 90.000 – 100.000 rendelés fog jönni
  - Nem ellenőrzünk minden rendelés előtt, hogy van-e biztosan még
  - Mindenkinek visszaigazoljuk azonnal, hogy megkaptuk a rendelést
  - Elmentjük a rendeléseket egy várósorba
  - Később elkezdjük ténylegesen feldolgozni a rendeléseket

- Megoldás: várósorba mentés → Event Store
- Technika
  - Redis
  - RabbitMQ
  - MQTT
  - HiveMQ
- Ezek az adatbázisok arra vannak optimalizálva, hogy villámgyorsan képesek legyenek elmenti kéréseket, nagyságrendekkel gyorsabban, mint egy relációs adatbázis
  - Előző szenzoros problémafelvetést is megoldja
  - Majd a valós szerverek aszinkron módon kiszedegetik innen a beérkezett kéréseket

# CQRS (+ Event sourcing, DB szétvágás nélkül)

17



- **Előnyei**

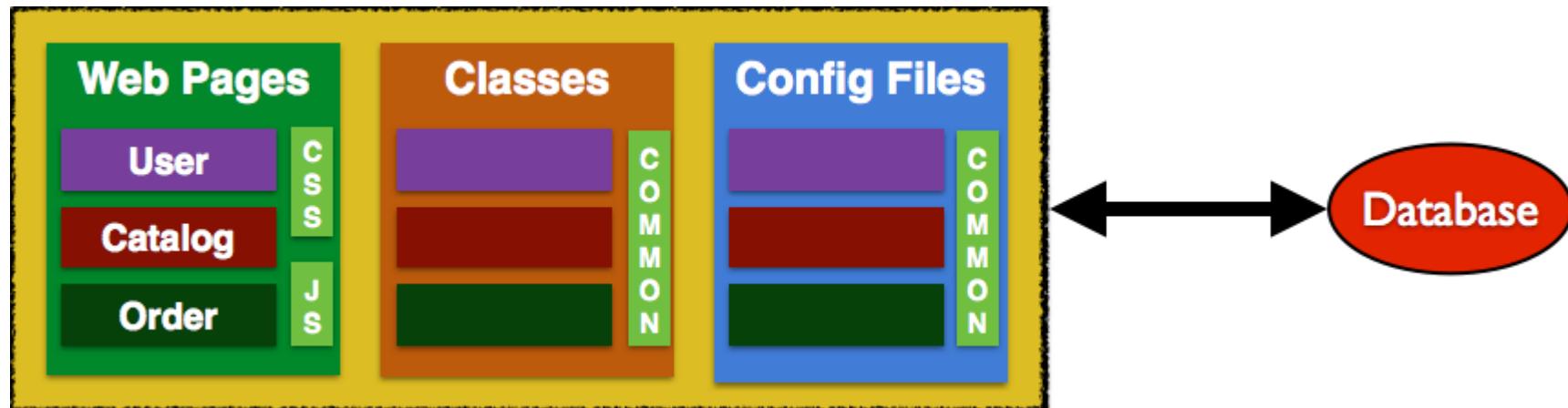
- Nagy teljesítmény
- Egyszerűbb a rendszerek összeépítése
- Könnyű hibakeresés, tesztelés
- Event Store-ból extra üzleti adat is kinyerhető

- **Hátrányai**

- Reporting bonyolult
- Nagyobb tárigény → Persistent View Model eltárolása
- Hibás kérések visszajelzése nem azonnali

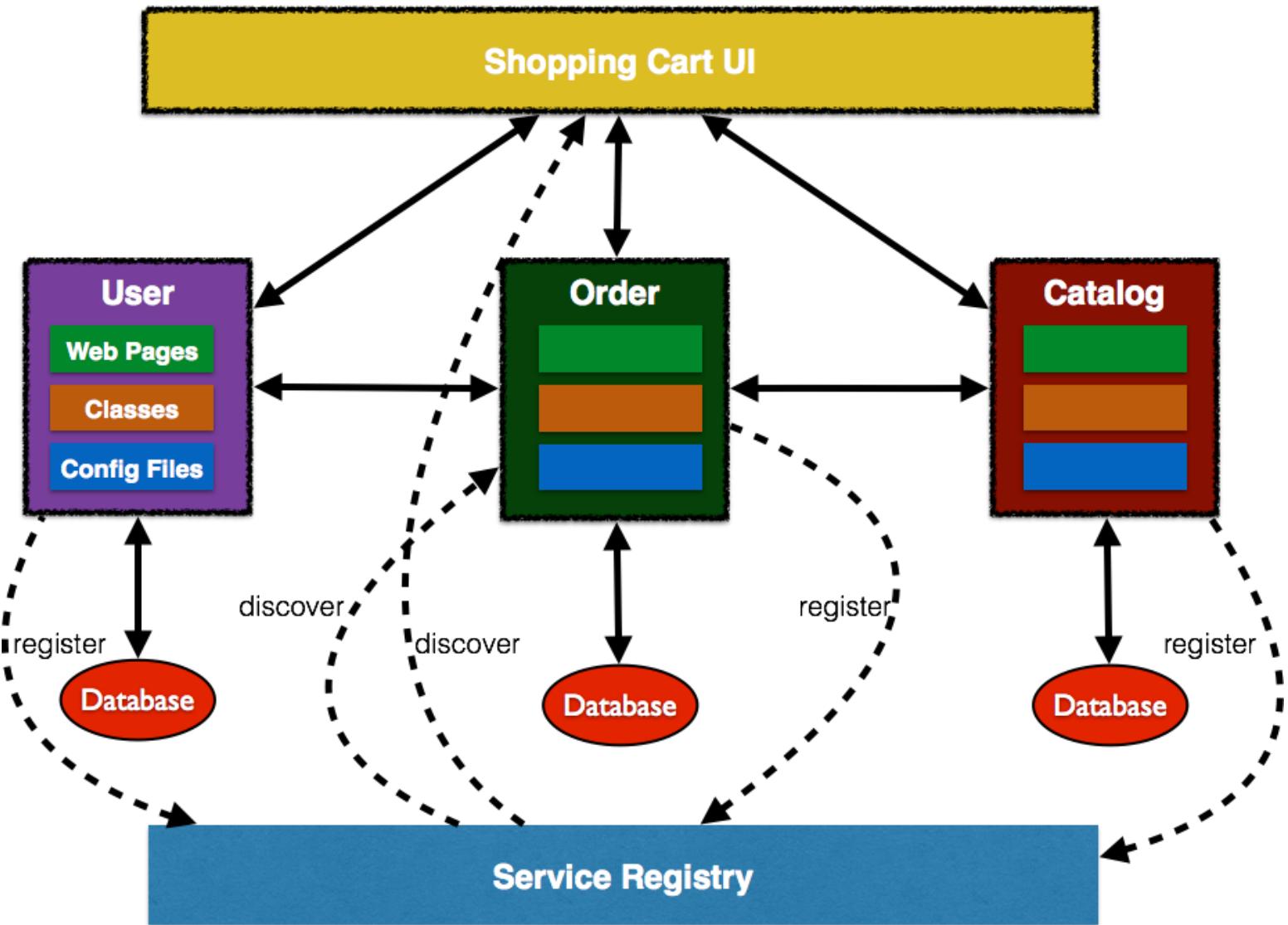
# Monolitikus alkalmazás

19



# Mikroszolgáltatások

20



- **Jellemzői**

- Önálló alkalmazások
- Felbontás alapja: domain model → bounded context
- Önálló fejlesztési ciklus
- Önállóan tesztelhetőek (service stubokkal)
- Önállóan skálázhatóak
- Akár más-más nyelven írhatóak
- Egymással valamilyen közösen ismert protokollon keresztül beszélgetnek
- Tipikusan konténerbe zárunk egy-egy mikroszolgáltatást

- **Belső működés**

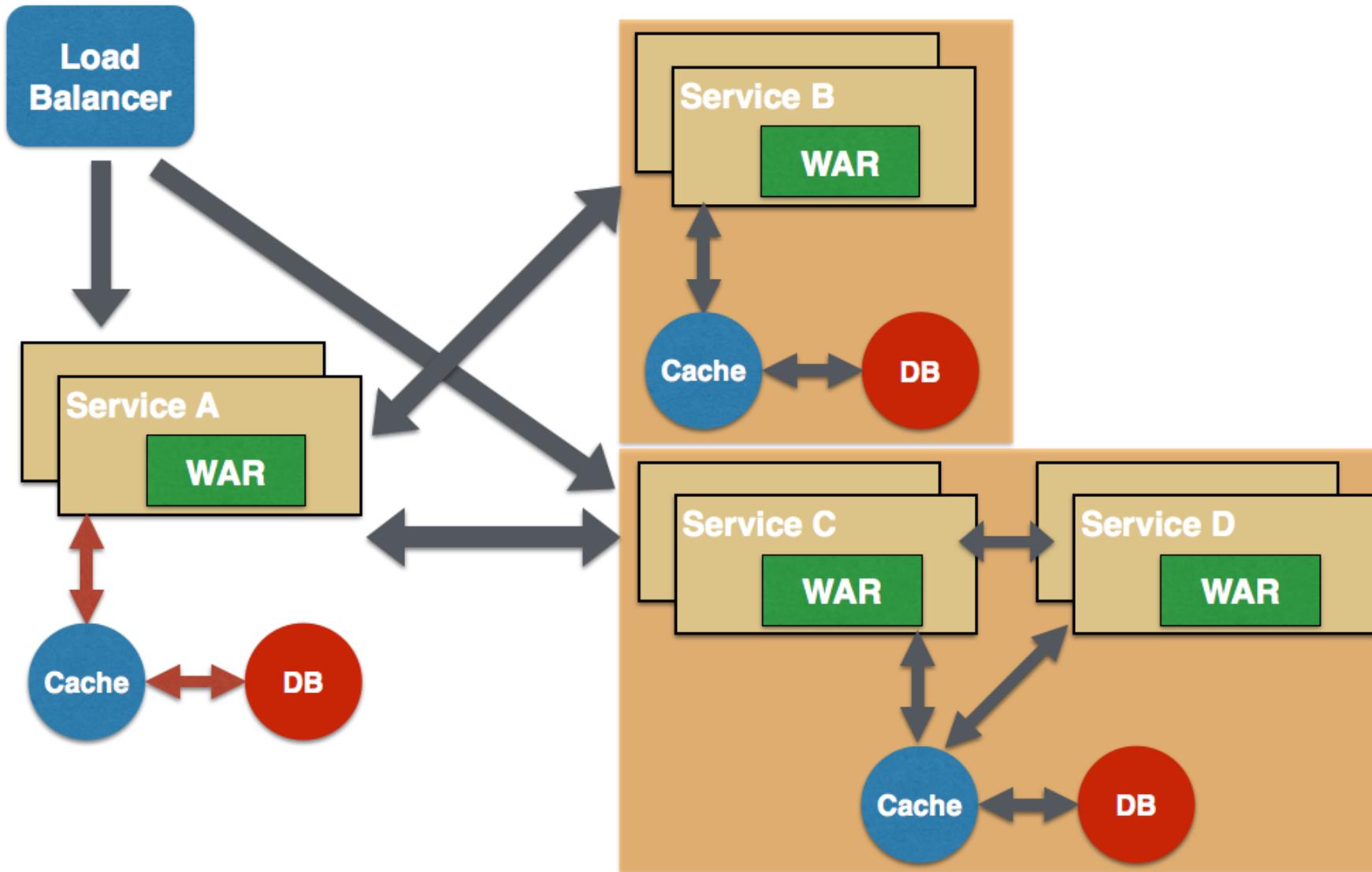
- Hagyományos rétegezéssel épül fel
- Saját adatbázissal rendelkezik/rendelkezhet
- Kommunikáció pl. REST API vagy Message bus protokollok

- **Külső elérés**

- Tipikusan REST API-n keresztül
- UI is egy mikroszolgáltatás, ami megjelenítésért felelős

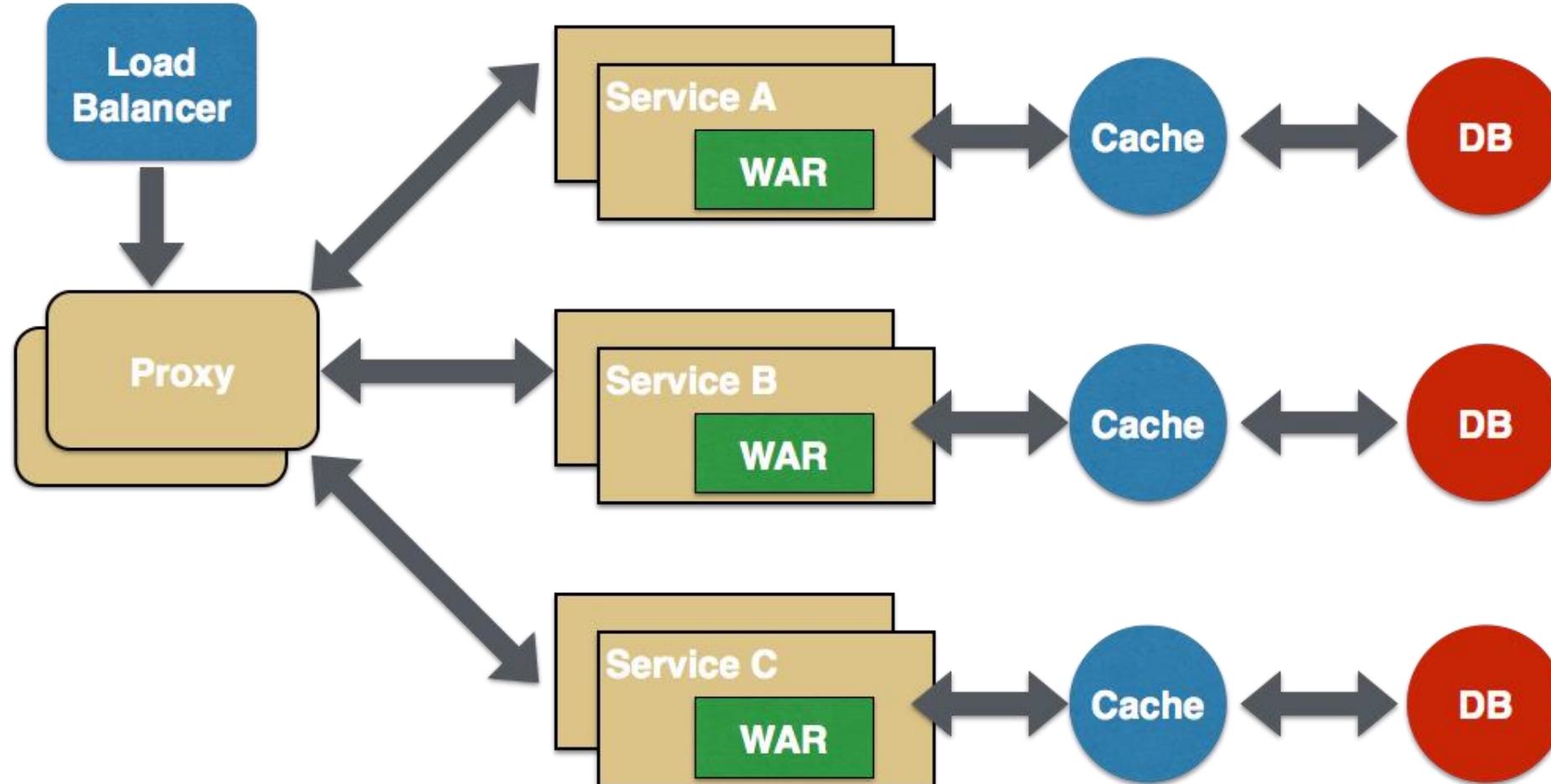
# Tervezési minták – nem strukturált

22



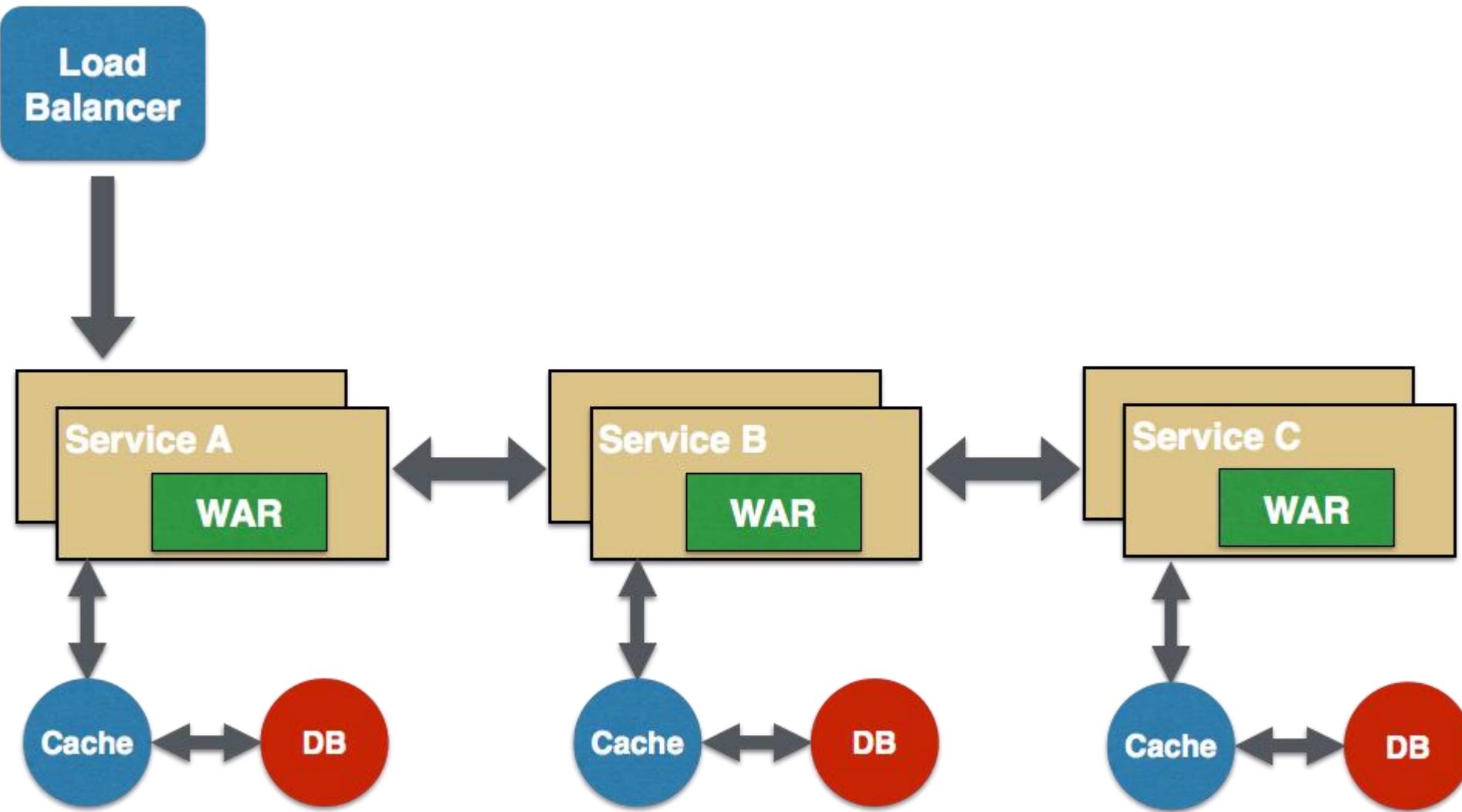
# Proxy/Facade

23



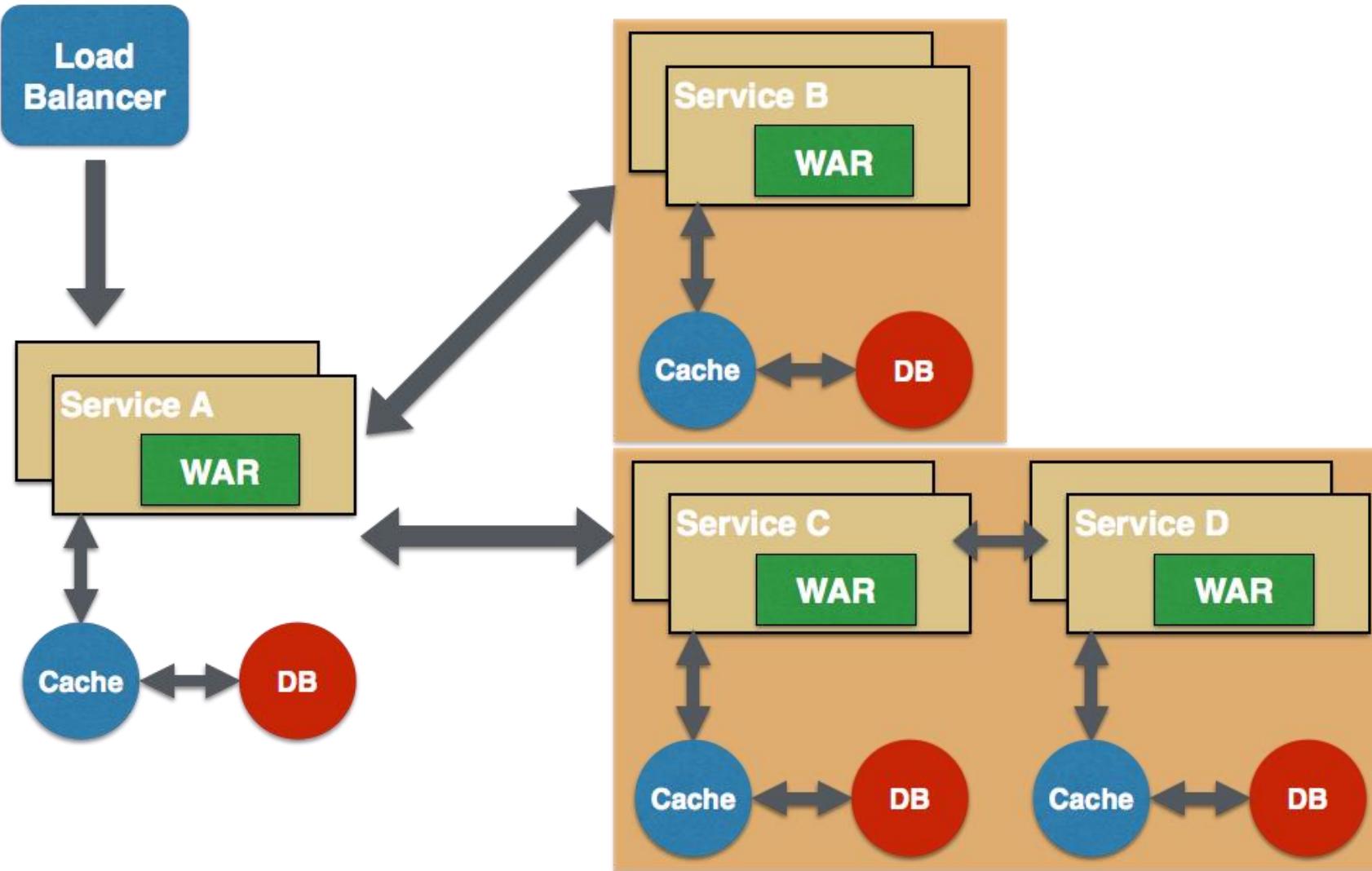
# Chain of Responsibility

24



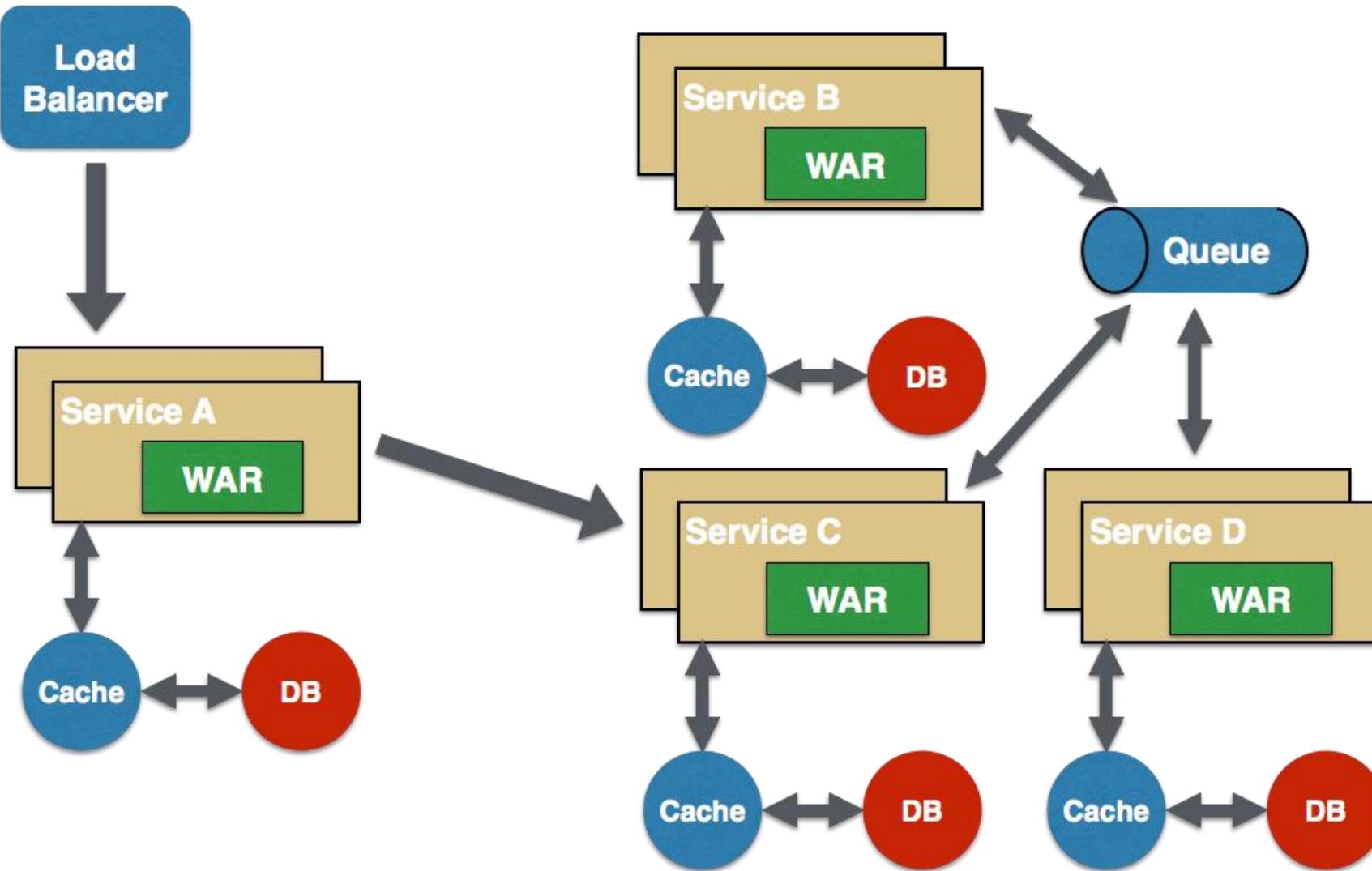
# Composite

25



# Mediator

26



- **Mediator technológia**

- **AMQP:** Advanced Message Queueing Protocol
  - Általános, nyílt protokoll
  - Tipikusan PC/WEB
- **MQTT:** Message Queue Telemetry Transport
  - ISO szabvány
  - Publish – Subscribe üzenetküldésre tökéletes
  - Kis overhead
  - Tipikusan mobil/IOT
  - Brokers
    - Mosquitto: 30k msg / sec
    - Moquette: 30-100k msg / sec
    - HiveMQ: 800k msg / sec
    - Redis: 1M msg / sec

# Köszönöm a figyelmet!

Kérdés esetén e-mailben szívesen állok rendelkezésre.