Szoftvertechnológia

ÓBUDAI EGYETEM
NEUMANN JÁNOS INFORMATIKAI KAR

MODUL 5

SOLID elvek ismétlése

Architekturális tervezési minták összefoglalása

GOF

Létrehozási tervezési minták

Gyors szoftverfejlesztés

- **Kizárólag 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

Megvalósítása

- 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 témában
  - 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)

SOLID elvek

- **Single Responsibility**
  - Minden osztály egy dologért legyen felelős (és azt jól lássa el)
  - Ha nem követjük, akkor
    - Spagetti 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 bővíthetőek
    - Tünet: leszármazott megírásakor módosítanunk kell az ősosztályt is (TILOS, frameworknél pedig lehetetlen)
    - Tünet: egy kis funkció hozzáadásakor több osztályt kell hozzáadni ugyanabban a hierarchiában

SOLID elvek

- **Liskov substitutable**
  - Ősosztály helyett utódpéldány legyen mindig használható
  - Compiler supported, hiszen OOP elv (polimorfizmus)
  - Ha egy kliensosztály eddig X osztállyal dolgozott, akkor tudnia kell X leszármazottjával is dolgoznia
  - 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 seggregation**
  - Sok kis interfészt használjunk egy hatalmas mindent előíró interfész helyett
  - Ha nem követjük, akkor
    - Tünet: Egy osztályt létrehozunk valamilyen célból, megvalósítjuk az interfészt és rengeteg üres, fölösleges metódusunk lesz
    - Tünet: Az interfészhez több implementáló osztály jön létre a kód legkülönbözőbb helyein, más-más részfunkcionalitással

SOLID elvek

- **Dependency Inversion**
  - A függőségeket ne az őket 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 klienst

MVC workflow

CONTROLLER

VIEW

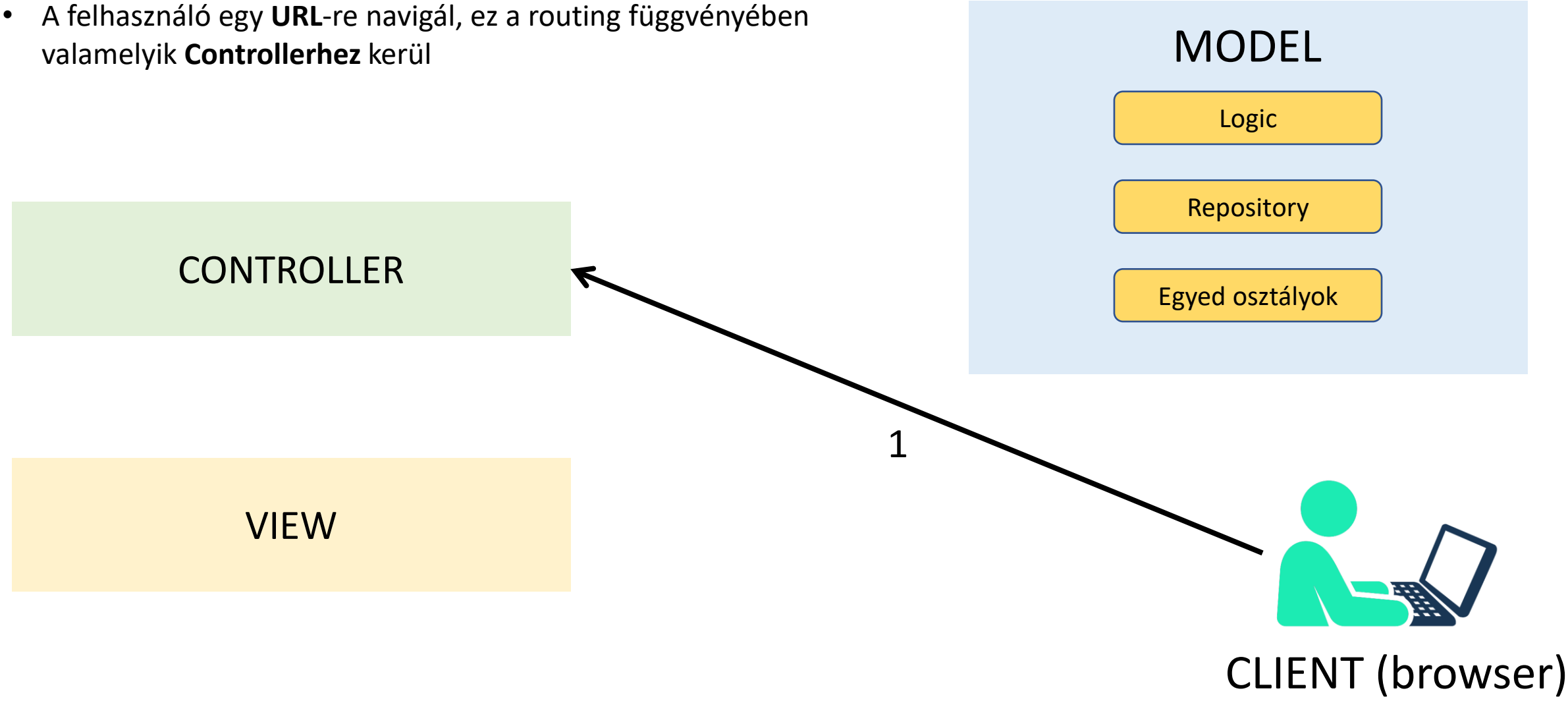
MODEL

- Logic
- Repository
- Egyed osztályok

CLIENT (browser)

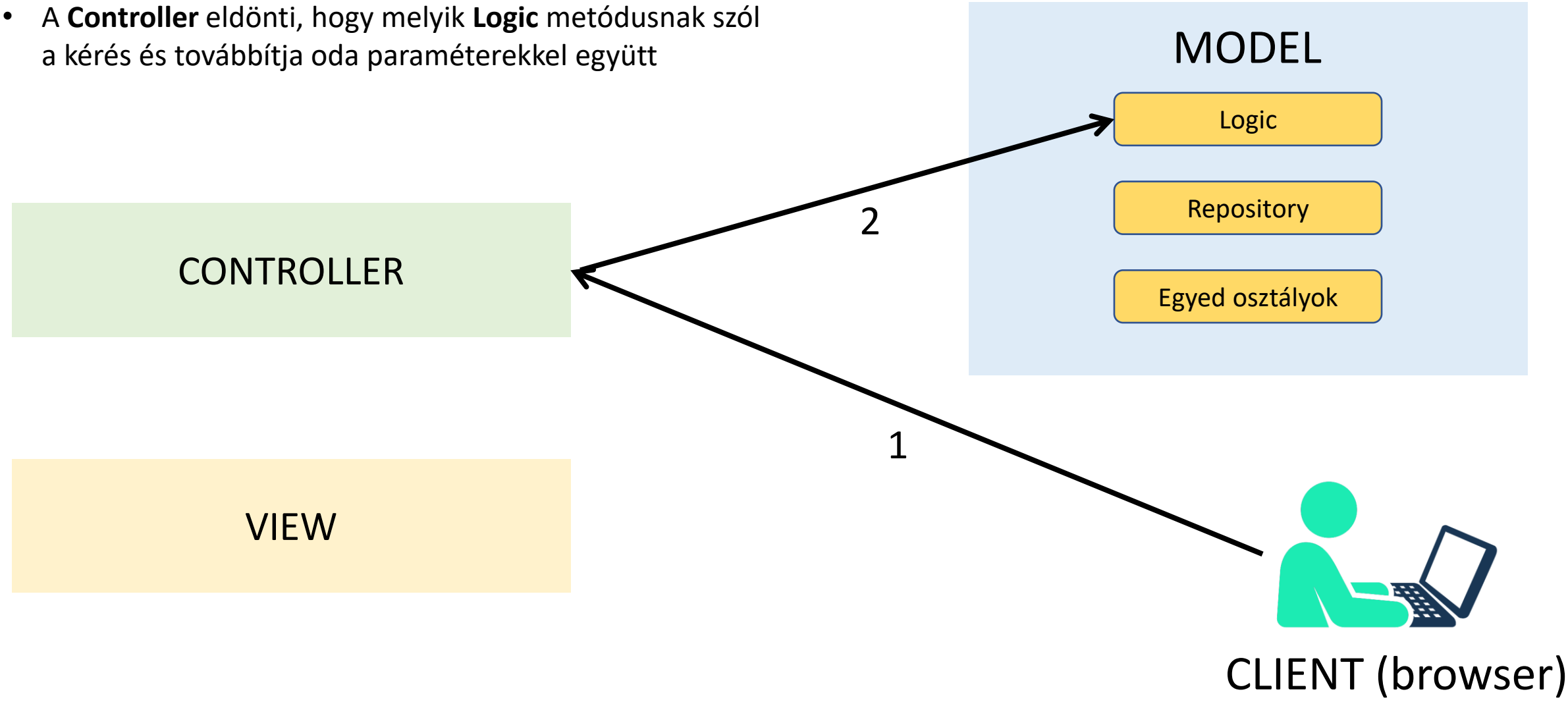
MVC workflow

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



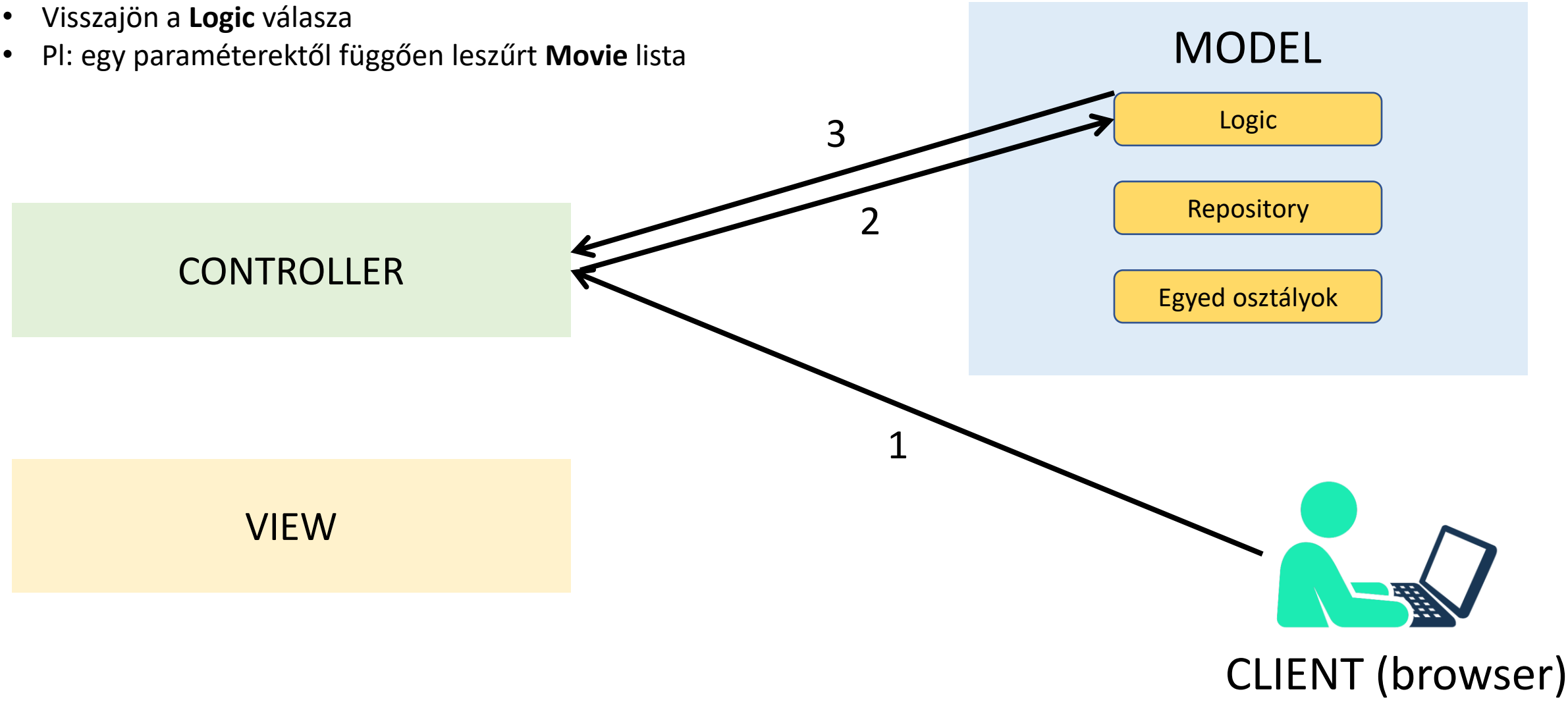
MVC workflow

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



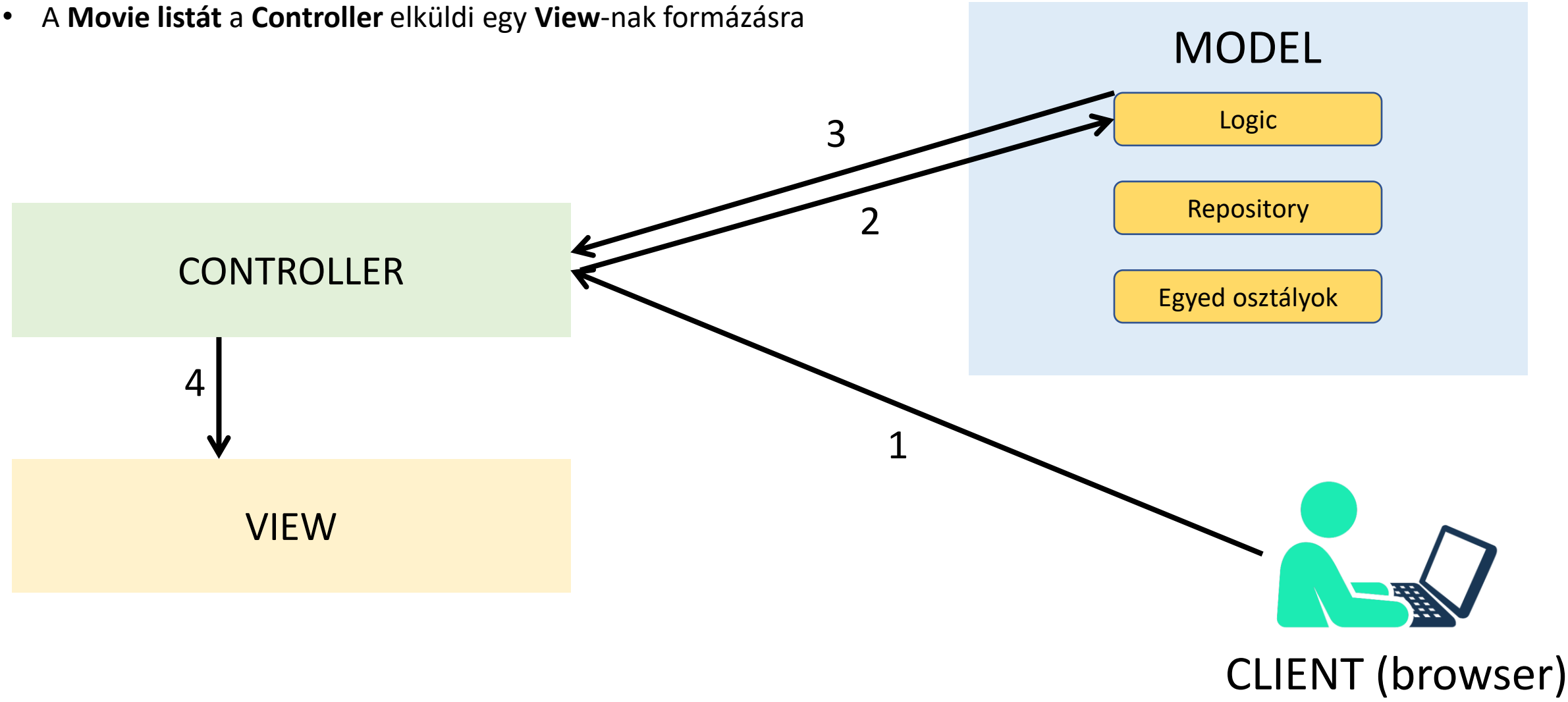
MVC workflow

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



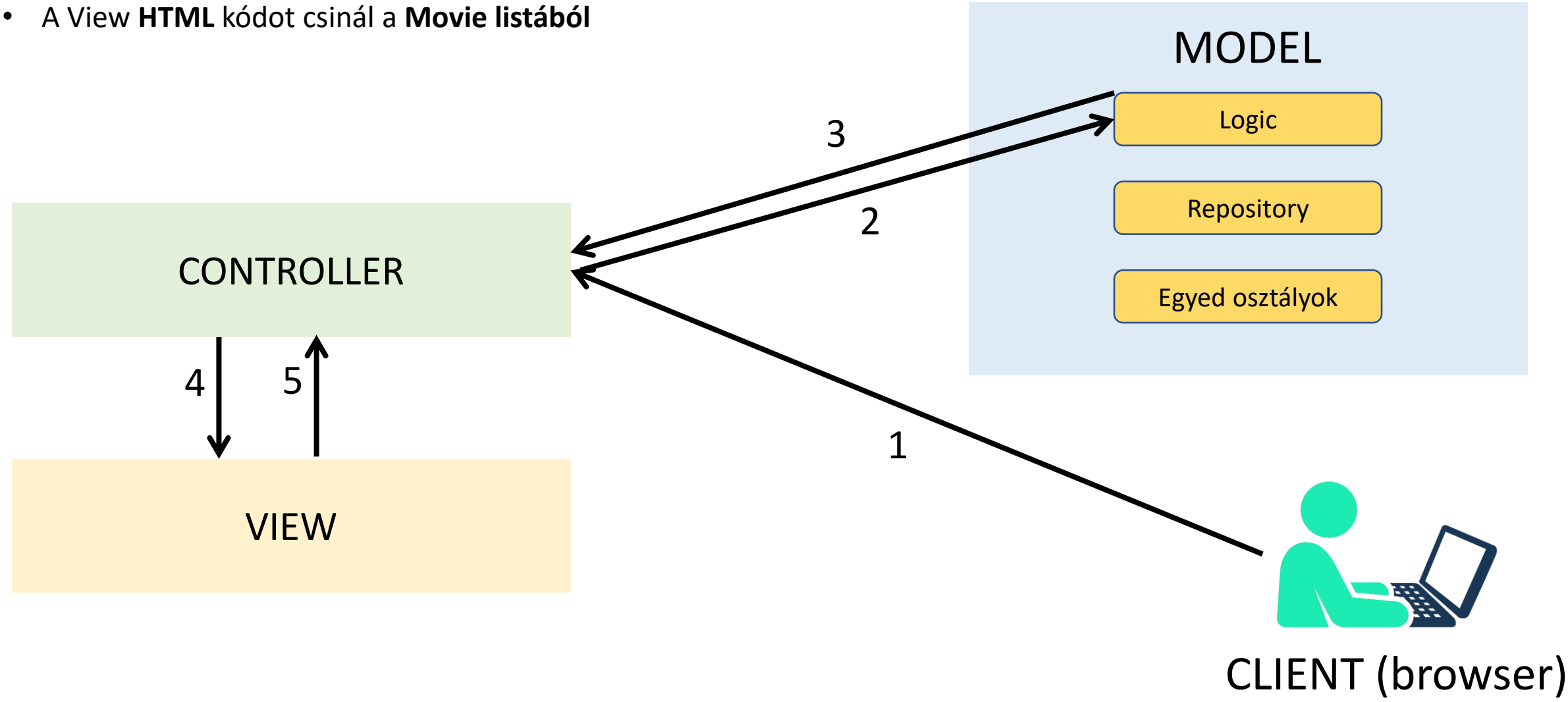
MVC workflow

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



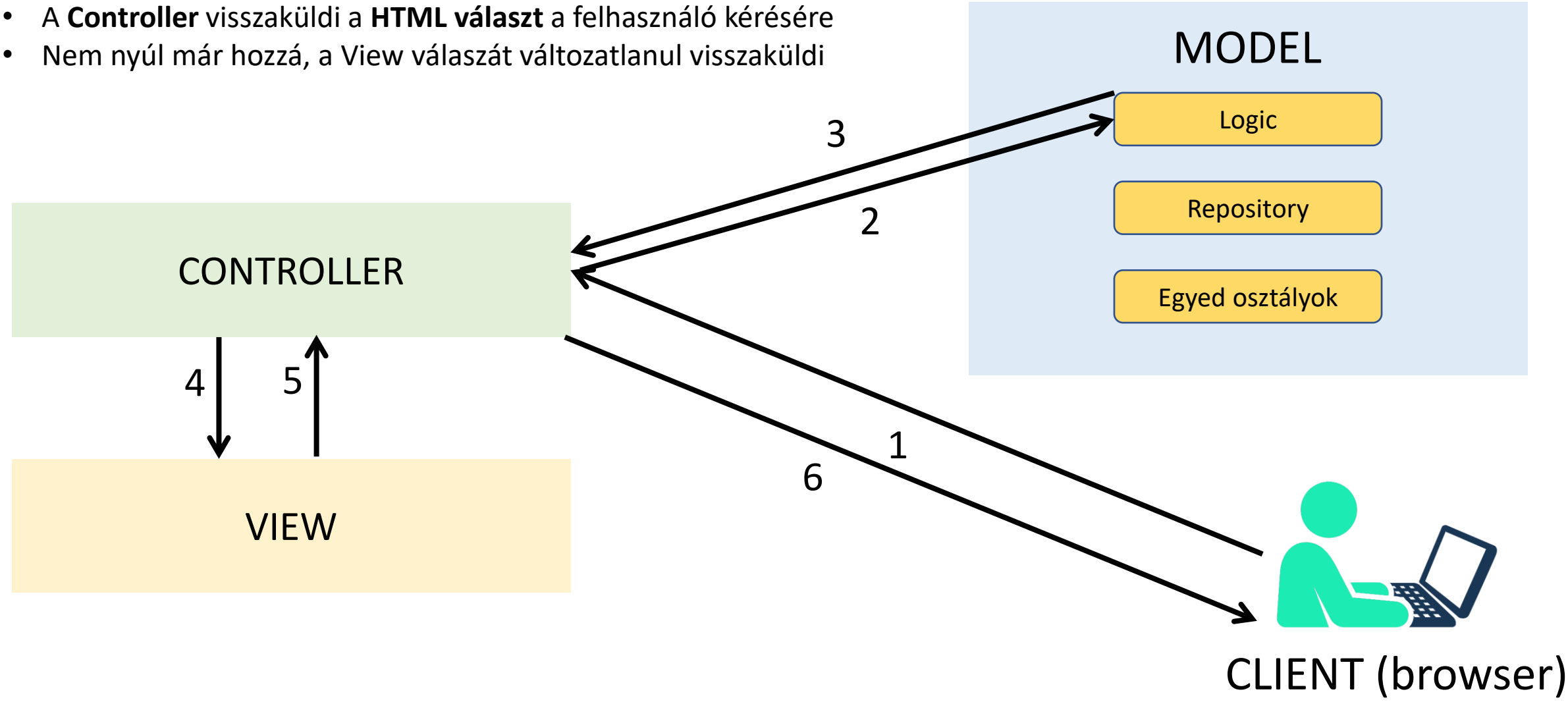
MVC workflow

- A View **HTML** kódot csinál a **Movie listából**



MVC workflow

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

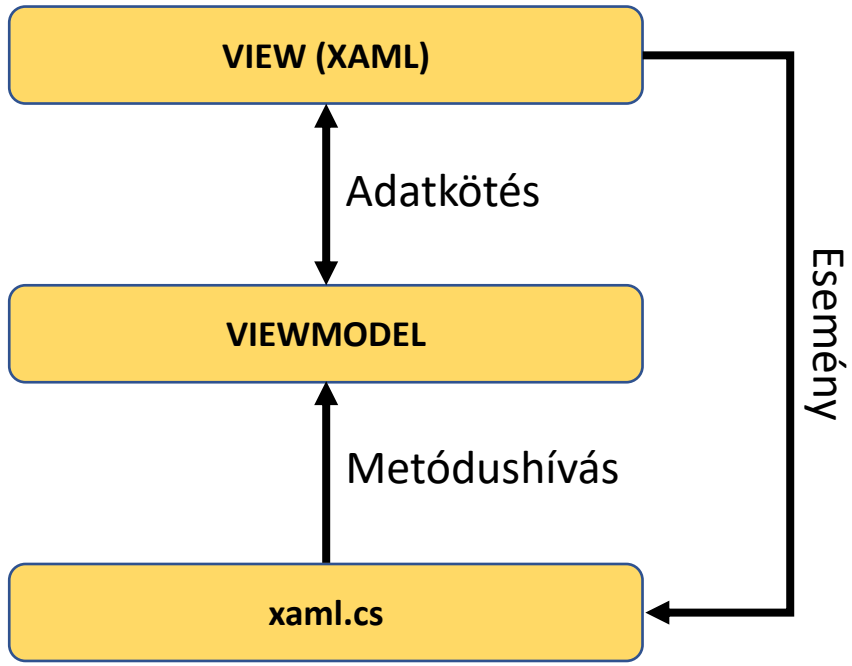


Hamis MVVM

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

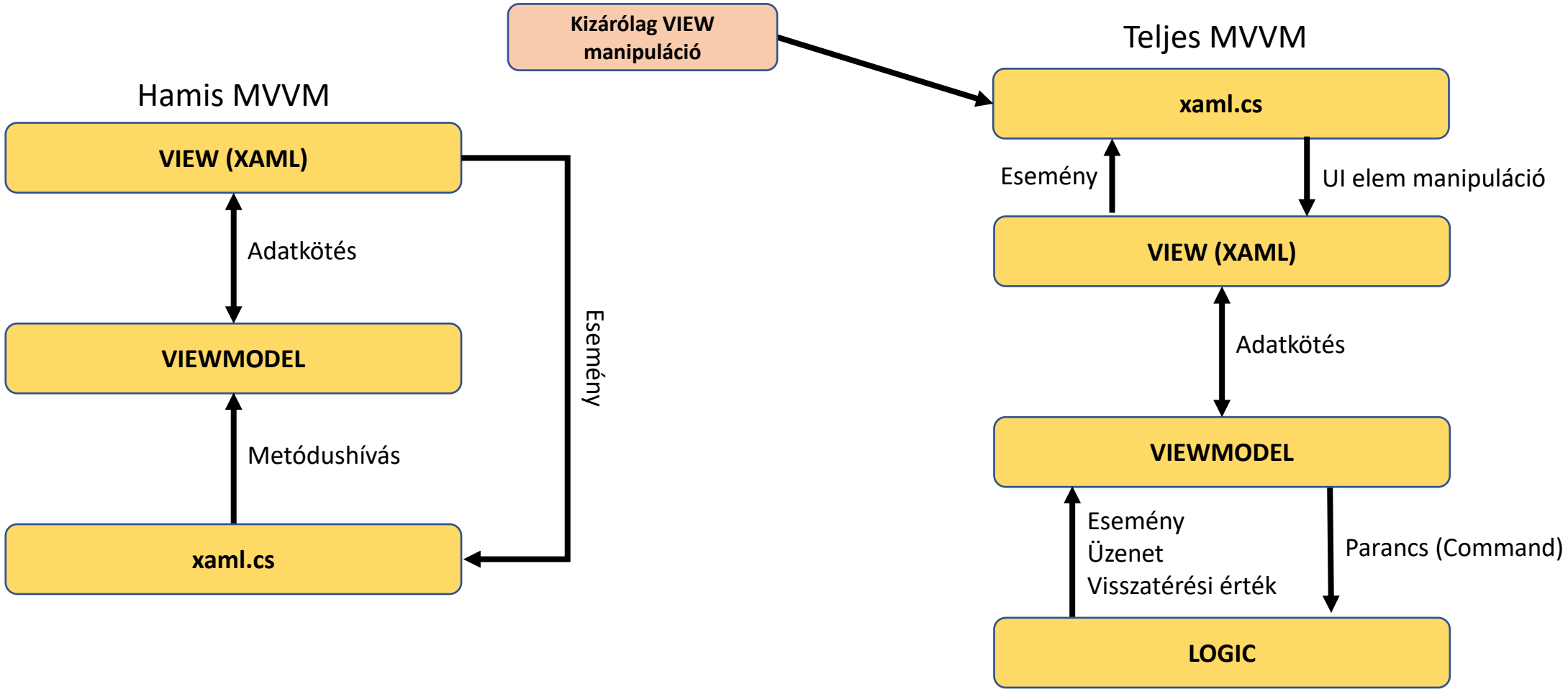
```
<Window.DataContext>
        <local:ViewModel />
</Window.DataContext>
```

  - Gyűjtemények → **ObservableCollection<T>**
  - Entitások → **INotifyPropertyChanged** megvalósítás
  - Metódusok → Eseményekből hívva
  - Tulajdonságok → UI vezérlőkre kötve



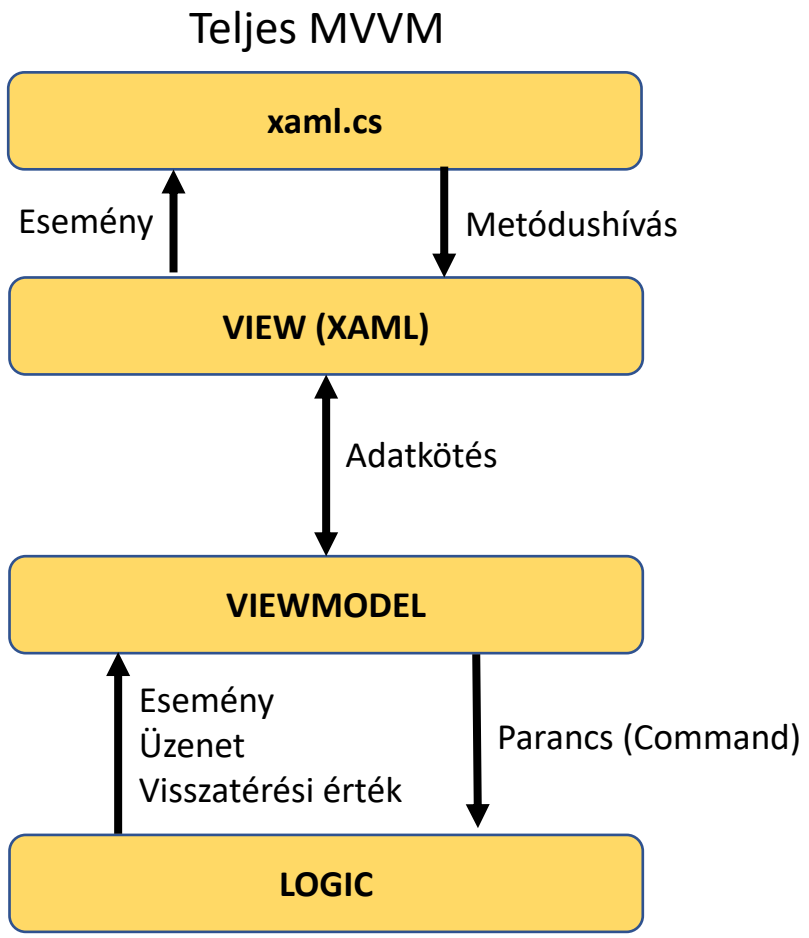
- Ez még nem MVVM pattern
  - Ez egyelőre a logika leválasztása
  - UI függ a logikától még → hiszen az eseménykezelőkben név szerint metódusokat hívunk

MVVM


Kizárólag VIEW manipuláció
Hamis MVVM
VIEW (XAML)
Adatkötés
VIEWMODEL
Metódushívás
xaml.cs
Esemény
Teljes MVVM
xaml.cs
Esemény
UI elem manipuláció
VIEW (XAML)
Adatkötés
VIEWMODEL
Esemény
Üzenet
Visszatérési érték
Parancs (Command)
LOGIC

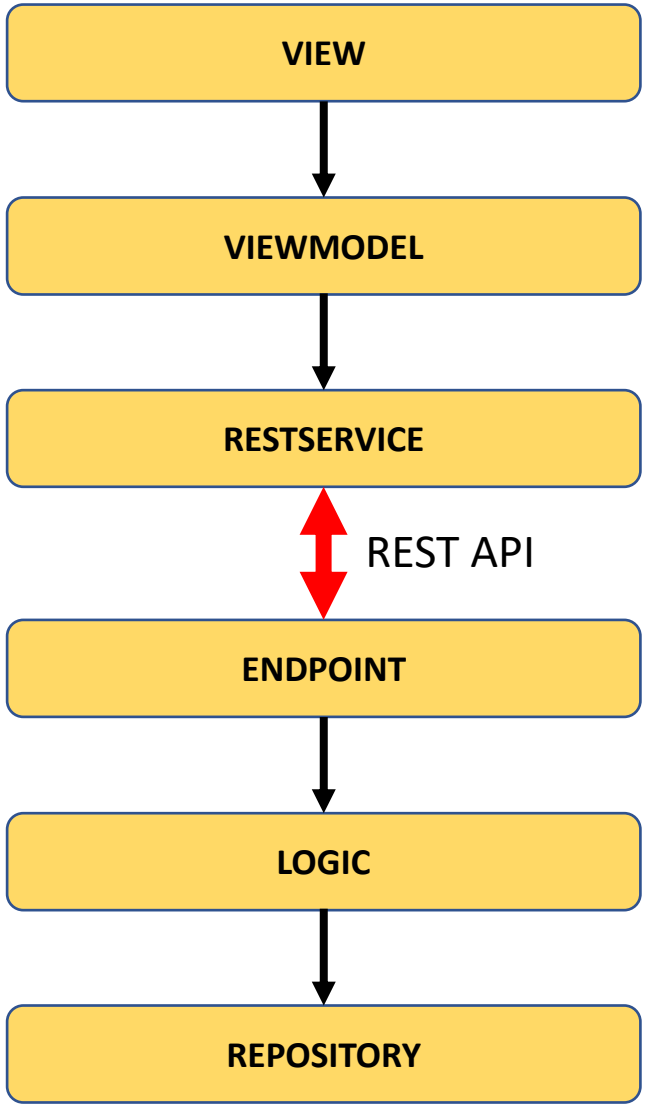
MVVM célja

- **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



MVVMC v2

- 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 minták

- **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 mindent a modern programozási nyelvekre kell formálni belőle
- Jellemzőik
  - Újrafelhasználás maximális kiszolgálása
  - Öröklésre vagy kompozícióra építenek

Refaktorálás

- **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

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

Factory Method

- **Probléma**: Az objektumainkat gyakran bonyolult létrehozni és a konstruktor 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 **Point**ot 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

Builder

- **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

Singleton

- **Probléma**
  - Jól jönne egy adott helyzetben statikus osztály használata, mert mindig ugyanaz az állapottér kell
  - Nem szeretnénk viszont statikus osztályt készíteni, mert nem tesztelhető
  - Kellene egy olyan megoldás, hogy egy osztálytól mindig 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

Prototype

- **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 eléggé összetett, van sok érték és sok referencia típus benne
- **Megoldás**
  - Osztályok szintjén Deep Copy metódusok létrehozása
  - Vagy szerializáció, majd deszerializáció

Köszönöm a figyelmet!

Kérdés esetén e-mailben szívesen állok rendelkezésre.