Haladó Fejlesztési Technikák

ÓBUDAI EGYETEM
NEUMANN JÁNOS INFORMATIKAI KAR

Modul 5

Rétegzés

SOLID elvek

Dependency Inversion

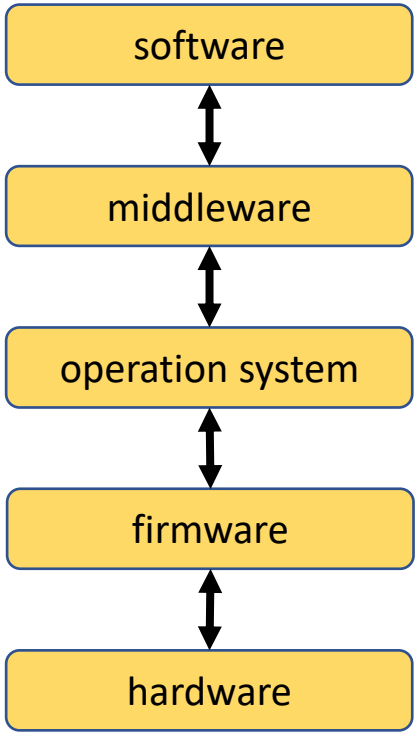
Repository Layer

Logic Layer

UI

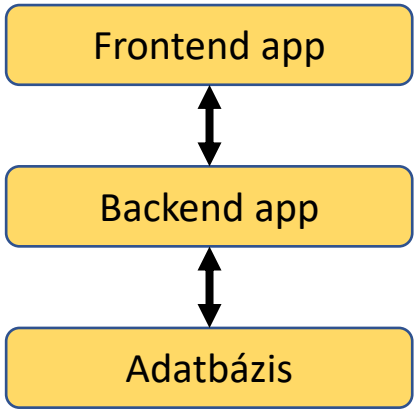
Rétegek

- **Cél:** az implementációs részletek elrejtése, hogy a felső réteg ne függjön a nem szomszédos alsó rétegektől
- Az operációs rendszer is így működik
- Minden réteg csak az alatta lévő réteget ismeri
- A C# szoftverünknek mindegy, hogy:
  - Milyen operációs rendszeren fut
  - Mindegy, hogy milyen processzoron fut
  - Csak .NET futtató környezet legyen alatta

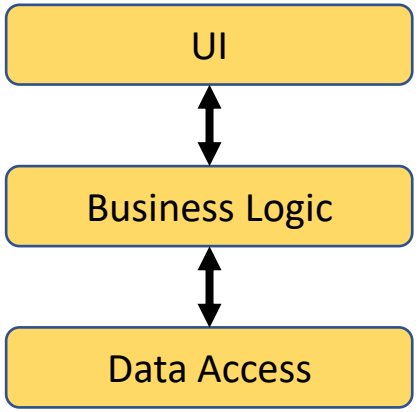


Tier vs Layer

- **Tier:** fizikai felbontása az alkalmazás komponenseknek
  - Szoftver vagy hardverkomponensek, amik a rétegeket futtatják
  - Nem feltétlenül azonos felbontású, mint a rétegek

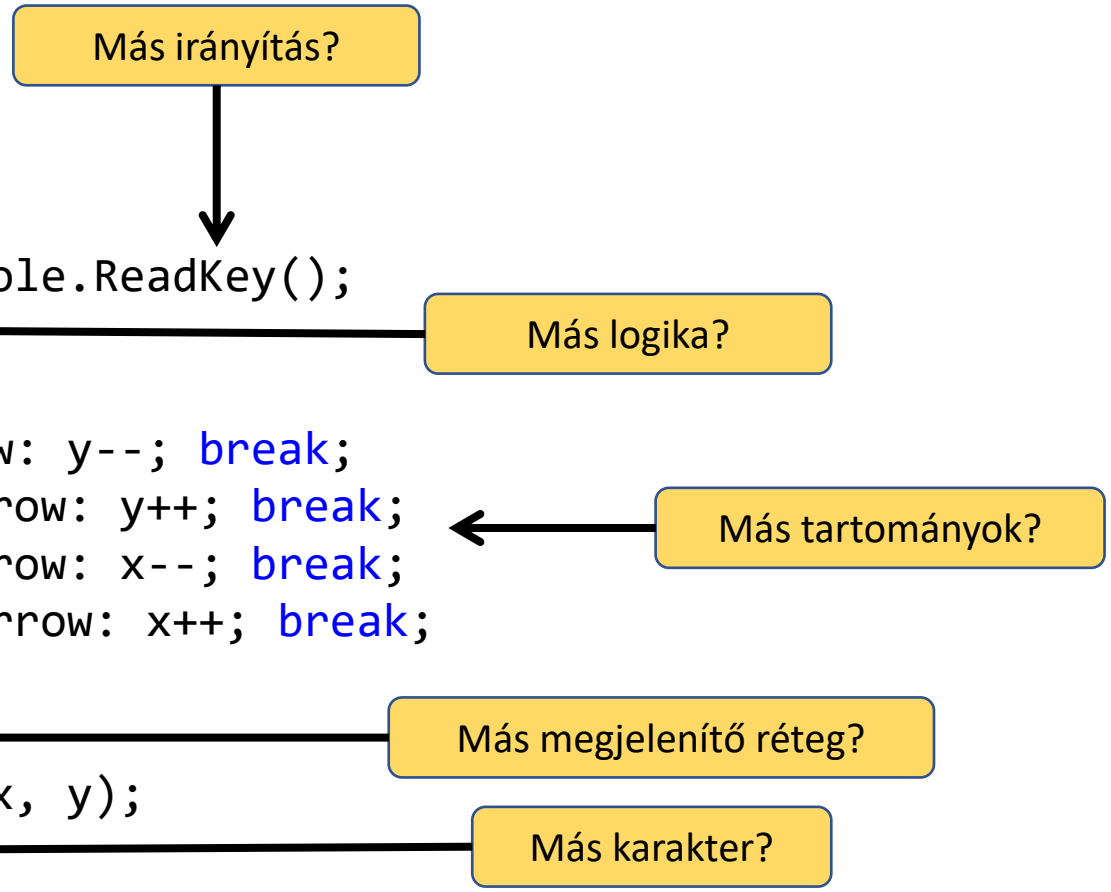


- **Layer**: logikai felbontása az osztályoknak
  - Az egy rétegbe tartozó osztályok egy közös, magasabb rendű feladatot látnak el
  - Egyértelműen definiált, hogy mi érthető el kívülről
  - Rétegek között interfészekkel teremtünk kapcsolatot
  - Akkor jó egy alkalmazás, ha egy réteg cserélhető egy máshogy implementált de azonos interfészekkel dolgozó másik komponensre



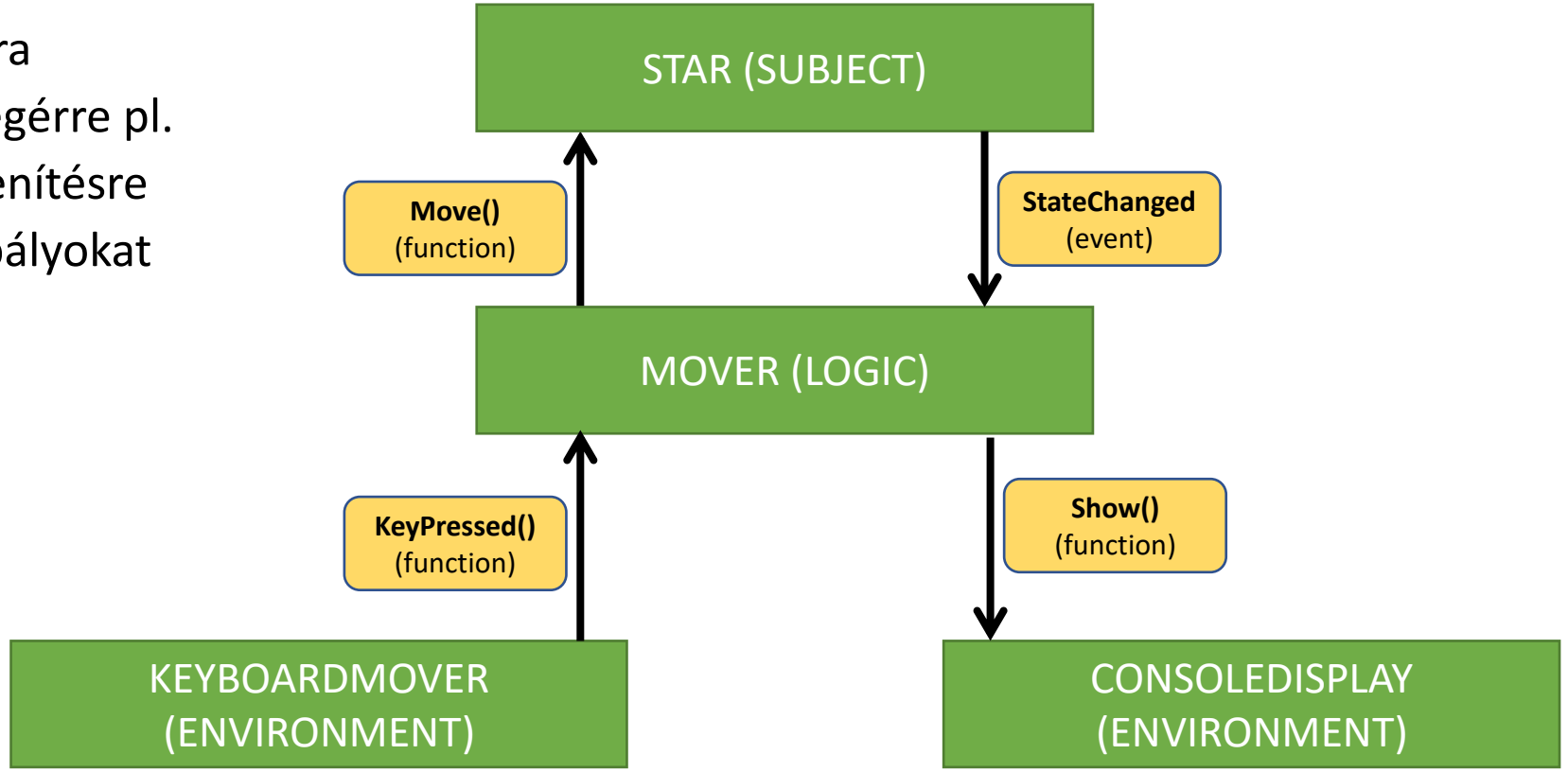
Nem rétegzett kód


```
int x = 0;
int y = 0;
while (true)
{
    ConsoleKeyInfo info = Console.ReadKey();
    switch (info.Key)
    {
        case ConsoleKey.UpArrow: y--; break;
        case ConsoleKey.DownArrow: y++; break;
        case ConsoleKey.LeftArrow: x--; break;
        case ConsoleKey.RightArrow: x++; break;
    }
    Console.Clear();
    Console.SetCursorPosition(x, y);
    Console.Write('*');
}
```



Rétegzett/fenntartható megoldás

- **Fenntartható**: Könnyen továbbfejleszthető, átültethető más környezetbe
- **Lehetőség van:**
  - Kicserélni a STAR-t másra
  - Áttérni billentyűzetről egérre pl.
  - Áttérni grafikus megjelenítésre
  - Átírni könnyedén a szabályokat



SOLID elvek

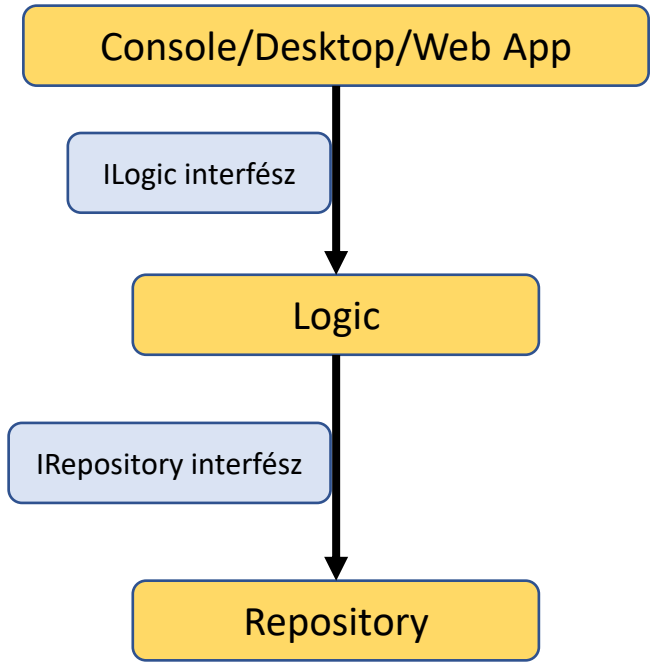
- **S – Single Responsibility**
  - Egy osztály – egy felelősségi kör elve
  - Ne legyen olyan osztály, ami adatot kezel, megjelenít, átalakít, adatbázist/fájlt ír
- **O – Open/Closed Principle**
  - Egy osztály ZÁRT a módosításra és NYITOTT a bővítésre
  - Vagyis ne írjunk át kódot egy meglévő osztályban, leszármazottakkal bővítsük
- **L – Liskov Substitution**
  - Ős osztály helyett bárhol a kódban legyen lehetőség leszármazottat használni hiba nélkül
- **I – Interface Segregation**
  - Sok kisebb interfész legyen egy nagy interfész helyett
- **D – Dependency Inversion**
  - Egy osztály ne függjön konkrét másik osztálytól → helyette interfésztől
  - A függőség létrehozása nem az adott osztály feladata

Dependency Inversion

- **Ne egy konkrét osztálytól, hanem egy funkcionalitástól függjünk**
  - A funkcionalitást bárki implementálhassa
- **Dependency Inversion megvalósítása**
  - Dependency Injection: interfész típusú konstruktor paraméterrel
  - Factory tervezési mintával (SZTGUI tárgy)
  - Inversion of Control (IoC) konténerrel (SZTGUI tárgy)

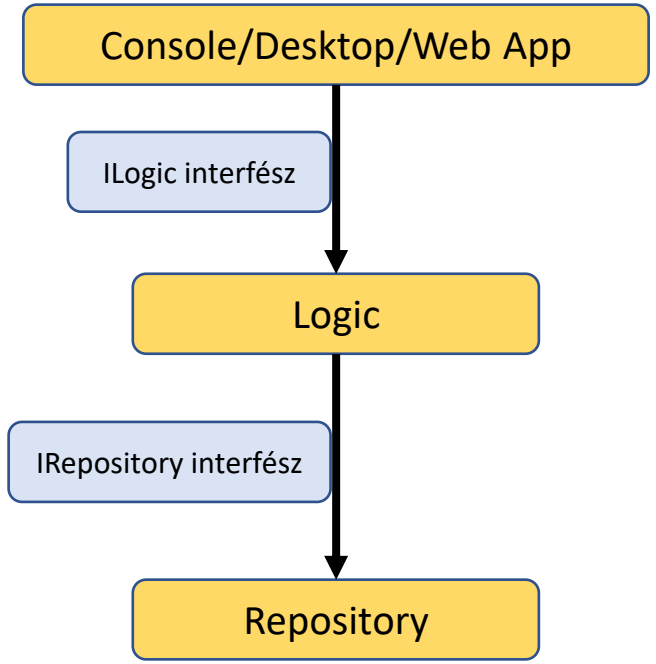
Rétegzett adatkezelő alkalmazás

- **Repository réteg**
  - A teljes adattárolási réteg kiszervezve
  - Entity Frameworkkel dolgozik
  - A **Logic** felé **CRUD** metódusokat biztosít:
    - Create, Read/ReadAll, Update, Delete
- **Logic réteg**
  - Az üzleti logikai kód helye (adattárolás nélkül)
  - A különböző entitásokat a **Repository** CRUD metódusaiból nyeri
  - Ezeket esetleg kibővítve továbbítja az alkalmazás felé
  - Biztosít ezeken felül bonyolult egyéb műveleteket
- **App réteg**
  - Példányosít egy **Logic**ot és használja a metódusait
  - Nincs tudomása a **Repository** létezéséről



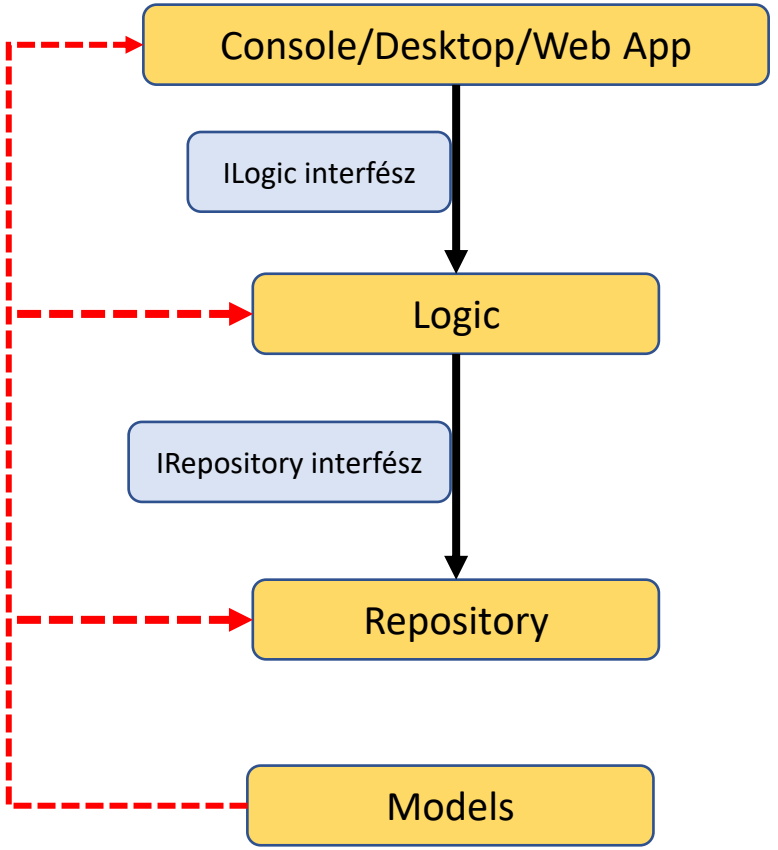
Rétegek C# megvalósítása

- **1 réteg = 1 projekt**
  - **Solution név**: MovieDb
    - **Models**: MovieDb.Models
    - **Repository**: MovieDb.Repository
    - **Logic**: MovieDb.Logic
    - **App**: MovieDb.Client
- **Projekt típusok**
  - **App**: Console App
  - **Többiek**: Class Library
- **Referenciák beállítása**
  - Project → jobb klikk → Add → Project reference
  - Mindenki az alatta lévőt ismerheti referenciaként
  - Models réteget mindenki ismerheti
  - Client mindenkit ismerhet az összeépítés miatt



MODELS réteg

- **Models**
  - Hivatalosan nem réteg
  - Egy DLL, amiben minden egyed osztályunk (pl: Car, Brand, Customer, stb.) benne van
  - Az egyed osztályokat minden rétegnek ismernie kell
  - Lesz vele probléma később (Entity → DTO konverzió kéne)
- **Interfészek**
  - Minden réteg között szükségesek
  - Mert „sosem függhetünk konkrét osztálytól”
  - **Cél:**
    - Bármikor a CarSQLRepository legyen cserélhető
    - Egy CarJSONRepository-ra
    - De azonos funkciókat teljesítsen: Create, Read, Update, Delete…
    - Logicot ugyan nem akarunk cserélni, de teszteléshez kell az interfész



Feladat

- Hozzuk létre az alkalmazásunk **MODELS** rétegét!

Repository réteg

- **Részei:**
  - DbContext leszármazott
  - IRepository interfész
  - Repository implementáció
- **IRepository interfész példa**

```
public interface IMovieRepository
{
        void Create(Movie movie);
        Movie Read(int id);
        IQueryable<Movie> ReadAll();
        void Update(Movie movie);
        void Delete(int id);
}
```

DbContext leszármazott

- **Checklist**
  - Publikus DbSet-ek létrehozása
  - **OnConfiguring()**
    - **UseSqlServer()** → MDF és LDF fájlok az adattáblák ellenőrzésére
    - **UseInMemoryDatabase()** → Ha a táblák jók, akkor így könnyebb a munka/beadás
    - **UseLazyLoadingProxies()** → Navigation Property-khez
  - **OnModelCreating()**
    - Idegen kulcsok beállítása Fluent API-val
    - DbSeed létrehozása

MovieRepository


```
public class MovieRepository : IMovieRepository
{
    MovieDbContext context;

    public MovieRepository(MovieDbContext context)
    {
        this.context = context;
    }

    public void Create(Movie movie)
    {
        this.context.Movies.Add(movie);
        this.context.SaveChanges();
    }

    public void Delete(int id)
    {
        this.context.Movies.Remove(Read(id));
        this.context.SaveChanges();
    }
}
```

- **MovieDbContext**
  - Nem interfészen keresztül érjük el
  - De nem baj, mert nem réteghatáron van
- **Create**
  - Hozzáadjuk az elemet
  - Mentjük az adatbázist
- **Delete**
  - A Read() segítségével megkeressük az elemet
  - Mentjük az adatbázist

MovieRepository


```
public Movie Read(int id)
{
        return this.context.Movies
                .FirstOrDefault(t => t.MovieId == id);
}

public IQueryable<Movie> ReadAll()
{
        return this.context.Movies;
}

public void Update(Movie movie)
{
        var oldmovie = Read(movie.MovieId);
        oldmovie.Income = movie.Income;
        oldmovie.Rating = movie.Rating;
        oldmovie.DirectorId = movie.DirectorId;
        oldmovie.Release = movie.Release;
        oldmovie.Title = movie.Title;
        this.context.SaveChanges();
}
```

- **Read**
  - Kikeressük az elemet id alapján
  - Ha nincs ilyen akkor:
    - **FirstOrDefault()** → null eredmény
    - **First()** → Exception
- **ReadAll**
  - Kötelezően IQueryable<T> visszatérés
  - Logicból szeretnénk még Linq-val szűrögetni
- **Update**
  - Nem szép megoldás
  - Megállapodunk benne, hogy módosítás után az ID a régi
  - Régi elemre másoljuk az új jellemzőket
  - Automatizálás: reflexióval, de lassú

Generikusság

- **Jelenleg**
  - 4 tábla esetén 8 osztály szükséges:
    - **4 db repository interfész**: IMovieRepository, IActorRepository, IDirectorRepository, IRoleRepository
    - **4 db repository implementáció**: MovieRepository, ActorRepository, DirectorRepository, RoleRepositoy
  - Ez roppant körülményes és ráadásul mindegyik implementáció ugyanazt csinálja
- **Új megoldás**
  - **Generikus ősinterfész**
    - Create(T item)
    - T Read(int id)
    - Stb.
  - **Absztrakt ősosztály**
    - Megoldja generikusan, amit meg tud
    - Amit nem, azt a konkrét implementációkra hagyja

Generikus megoldás


```
public abstract class Repository<T> : IRepository<T> where T : class
{
   protected MovieDbContext ctx;
   public Repository(MovieDbContext ctx)
   {
      this.ctx = ctx;
   }
   public void Create(T item)
   {
      ctx.Set<T>().Add(item);
      ctx.SaveChanges();
   }

   public IQueryable<T> ReadAll()
   {
      return ctx.Set<T>();
   }

   public void Delete(int id)
   {
      ctx.Set<T>().Remove(Read(id));
      ctx.SaveChanges();
   }

   public abstract T Read(int id);
   public abstract void Update(T item);
}
```

```
public interface IRepository<T> where T : class
{
           IQueryable<T> ReadAll();
           T Read(int id);
           void Create(T item);
           void Update(T item);
           void Delete(int id);
}
```

```
public class MovieRepository : Repository<Movie>, IRepository<Movie>
{
   public MovieRepository(MovieDbContext ctx) : base(ctx)
   { }

   public override Movie Read(int id)
   {
      return this.ctx.Movies.First(t => t.MovieId == id);
   }

   public override void Update(Movie item)
   {
      var old = Read(item.MovieId);
      foreach (var prop in old.GetType().GetProperties())
      {
         prop.SetValue(old, prop.GetValue(item));
      }
      ctx.SaveChanges();
   }
}
```

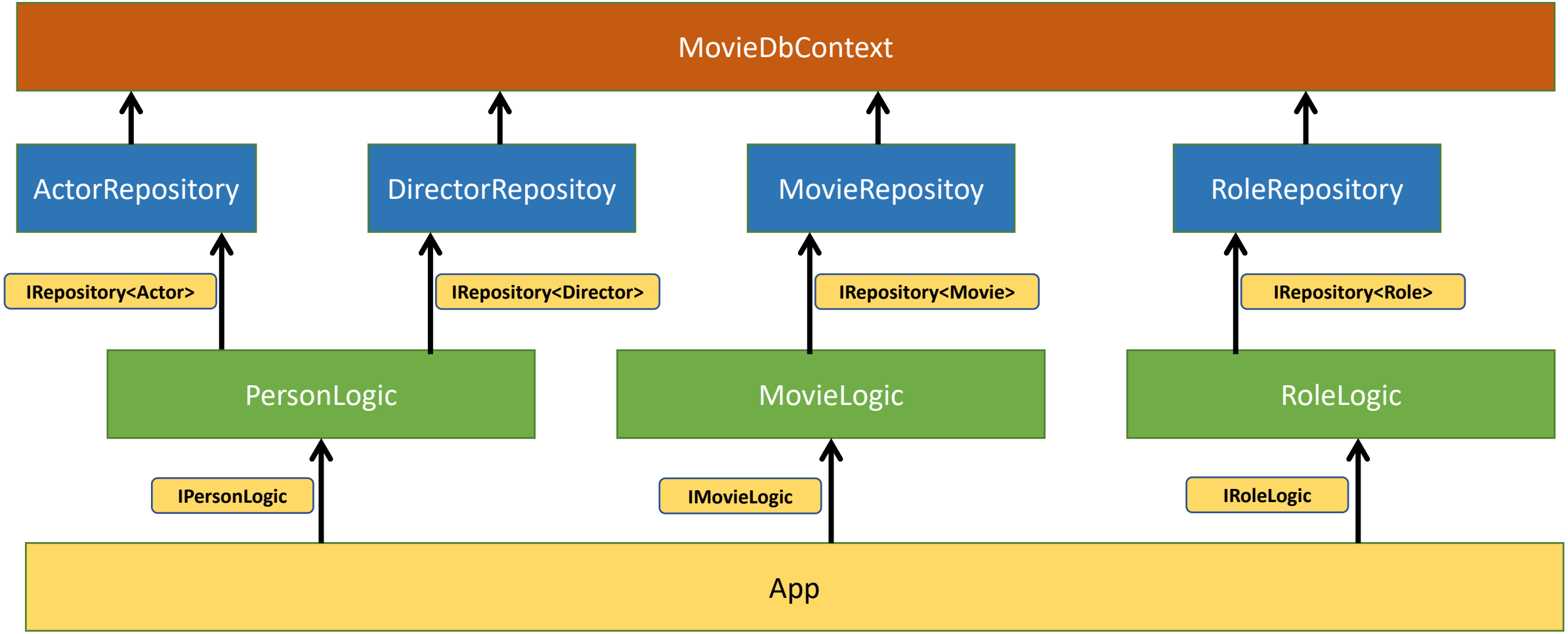
Feladat

- Hozzuk létre az alkalmazásunk **REPOSITORY** rétegét!

Logic

- **Szerepe**
  - Repository CRUD műveleteit továbbítja az App felé
    - De hozzájuk tehet, kiegészítheti őket
    - **Repository Create** nem csinál real life ellenőrzést (pl. autó ára nem lehet negatív), mert generikus…
    - A **Logic Create** szerepe
      - Ellenőrzéseket csinál, exception-öket dob
      - Sikeres ellenőrzés esetén az objektumot a **Repository Create**-hez továbbítja
  - Non-CRUD műveleteket biztosít
    - Bonyolult, többtáblás lekérdezések
    - Komplex kimutatások
- **Egyéb**
  - Ismerhet ős-repository interfészen át akár több konkrét repository-t (pl. Actor, Director)
  - Nem feltétlenül **egy repository** – **egy logic** a jó felosztás, sőt kerülendő

Logic felosztás példa



Felosztási gondok

- **Spaghetti code / Big Ball of Mud**
  - Monolitikus, átláthatatlan kód
- **Ravioli code**
  - Elméletileg kicsi és könnyen érthető osztályok
  - A rendszer egészének a megértése nehéz
- **Lasagne code**
  - Elméletileg és ránézésre rétegzett kód
  - Belső káosz és a szövevényes kereszthivatkozások miatt kezelhetetlen
- **Cél**
  - Aranyközépút megtalálálása
  - Nem baj, ha nem tökéletes a felosztás
  - Legyen fenntartható a kód

Logic implementáció


```
public class MovieLogic
{
   IRepository<Movie> repository;
   public MovieLogic(IRepository<Movie> repository)
   {
      this.repository = repository;
   }

   public void Create(Movie item)
   {
      if (item.Title.Length < 3)
      {
         throw new ArgumentException("Title too short!");
      }
      else
      {
         this.repository.Create(item);
      }
   }
   public Movie Read(int id)
   {
      var movie = this.repository.Read(id);
      if (movie == null)
      {
         throw new ArgumentException("Movie not exists");
      }
      return movie;
   }
}
```

- **Dependency Inversion**
  - Logic függ a Repository-tól
  - De csak interfészen át a funkcionalitástól
  - Nem ő maga példányosítja, kívülről várja
  - Konstruktor paraméteren át adja valaki kívülről
    - Ez a dependency injection
- **Kivételkezelés**
  - Create, Read a repository továbbításon kívül egyéb validációkat is végez
  - Pl. visszakapott null-ból itt lesz exception

Logic implementáció


```
public void Delete(int id)
{
   this.repository.Delete(id);
}

public IEnumerable<Movie> ReadAll()
{
   return this.repository.ReadAll();
}

public void Update(Movie item)
{
   this.repository.Update(item);
}

public double? GetAverageRatePerYear(int year)
{
   return this.repository
      .ReadAll()
      .Where(t => t.Release.Year == year)
      .Average(t => t.Rating);
}
```

- **IEnumerable<T>**
  - Itt már csak bejárható felületet adunk az app felé
  - Az app nem akar LINQ-zni rajta
- **Non-crud**
  - **GetAverageRatePerYear()** egy plusz funkció
  - A **Repository ReadAll()**-ján linq-zik
    - Az IQueryable<T>, tehát továbbítja a kérést
- **Továbbítások**
  - A **Delete(), ReadAll(), Update()** csak továbbítja a kérést a **Repository** felé

Non-CRUD metódusok

- Tipikusan valamilyen összetett lekérdezések
- Névtelen típusok használata helyett konkrét osztályok → vissza kell adni őket!

```
public IEnumerable<YearInfo> YearStatistics()
{
        return from x in this.repository.ReadAll()
                group x by x.Release.Year into g
                select new YearInfo()
                {
                        Year = g.Key,
                        AvgRating = g.Average(t => t.Rating),
                        MovieNumber = g.Count(),
                        RoleNumber = g.Sum(t => t.Roles.Count)
                };
}
```

```
public class YearInfo
{
   public int Year { get; set; }
   public double? AvgRating { get; set; }
   public int MovieNumber { get; set; }
   public int RoleNumber { get; set; }
}
```

Logic interfész

- **IMovieLogic**
  - Nem akarjuk cserélni, tehát a console app függhetne tőle akár
  - De jövő félévben a UI-tól le akarjuk választani teljesen
  - Teszteléshez is szükséges az interfész

```
public interface IMovieLogic
{
    void Create(Movie item);
    void Delete(int id);
    double? GetAverageRatePerYear(int year);
    Movie Read(int id);
    IEnumerable<Movie> ReadAll();
    void Update(Movie item);
}
```

Feladat

- Hozzuk létre az alkalmazásunk **LOGIC** rétegét!

Client réteg

- App réteg szabályok
  - Csak **Logic** műveleteket hív
  - Ezek eredményeit jelzi ki
  - **Javasolt**: ConsoleMenu-Simple nuget csomag → menügenerálás
- Minden példányosítás itt történik
  - DbContext, Repository, Logic
  - 1db **DbContext** példány létezhet → minden **repository** ezt kapja meg
  - Minden különböző **Repository**-ból csak 1db létezhet → minden **logic**, aminek kell, ezt kapja meg
- Egyéb szabályok
  - Csak a ConsoleApp használhat Console.Read/Write műveleteket
  - Logic és ConsoleApp nem használhat DbContext metódusokat
  - **Mindenki csak az alatta lévő réteg létezéséről tudhat**

Feladat

- Hozzuk létre az alkalmazásunk **CLIENT** rétegét!

Köszönöm a figyelmet!

Kérdés esetén e-mailben szívesen állok rendelkezésre.