

Haladó fejlesztési technikák

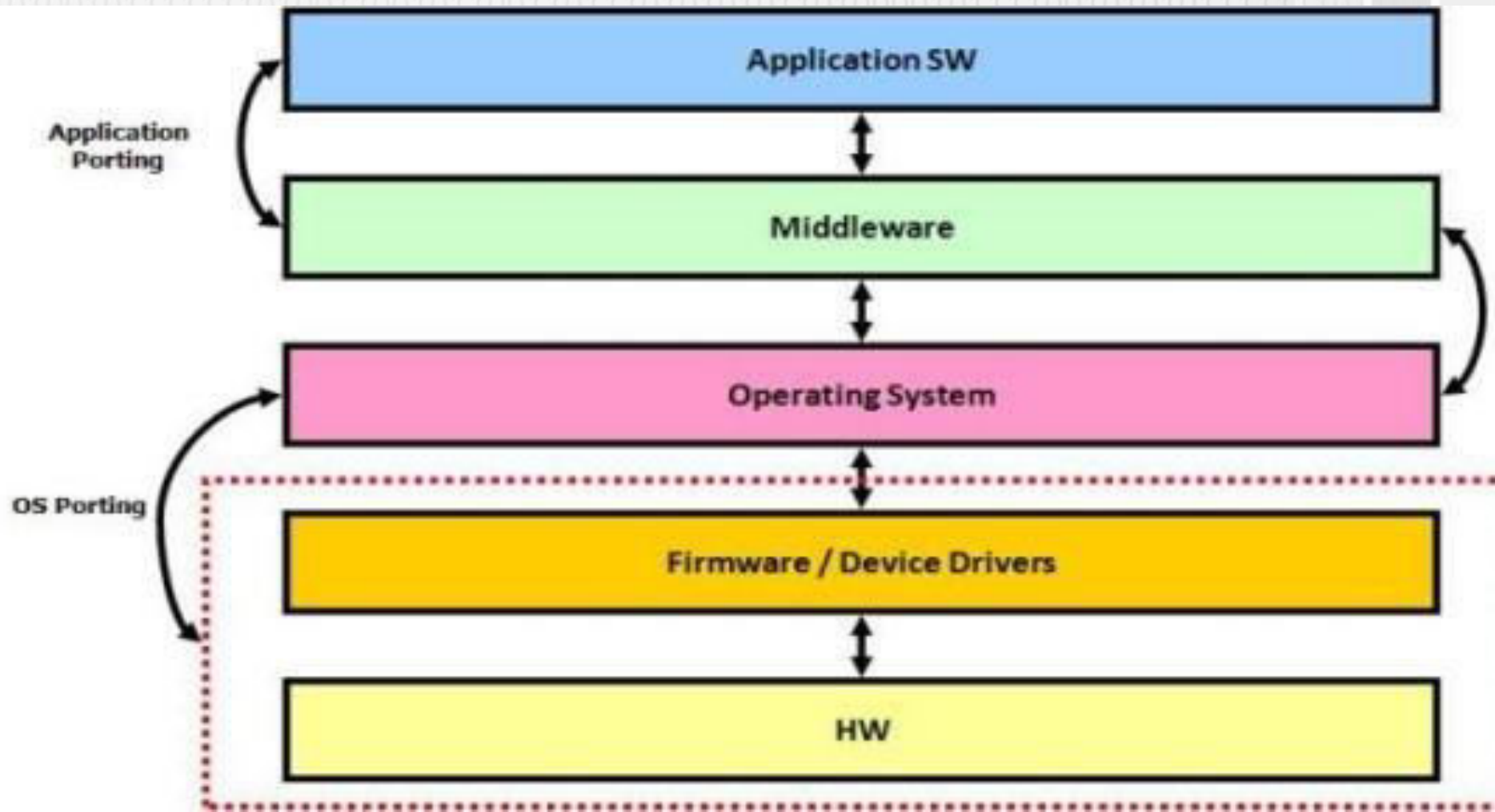
Layering, SOLID elvek

Adatbázisok elérése DbConnection/DbReader módszerrel

LocalDb elérése Entity Framework Core segítségével (ORM + LINQ)

Rétegek (layering)

- Cél: az implementációs részletek elrejtése, hogy a „felső” réteg ne függjön a nem szomszédos „alsó” rétegektől
- Hasonló elv: operációs rendszer rétegzett felépítése



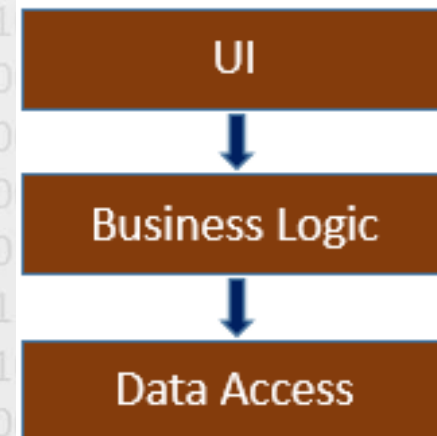
Tier vs Layer

- **Layer: logikai felbontása az osztályoknak**

- Az egy rétegbe tartozó osztályok egy közös „magasabb rendű” feladatot szolgálnak ki (pl. megjelenítés, adatkezelés...)
- Egyértelműen definiált, hogy melyek azok az osztályok, amelyek a rétegen kívülről elérhetőek
- A külső (szolgáltatott és felhasznált) funkciókat interfészekkel írjuk le
- Akkor jól megírt egy alkalmazás, ha egy réteg egy az egyben kicserélhető egy teljesen máshogy implementált, de azonos interfészekkel dolgozó másik komponensre

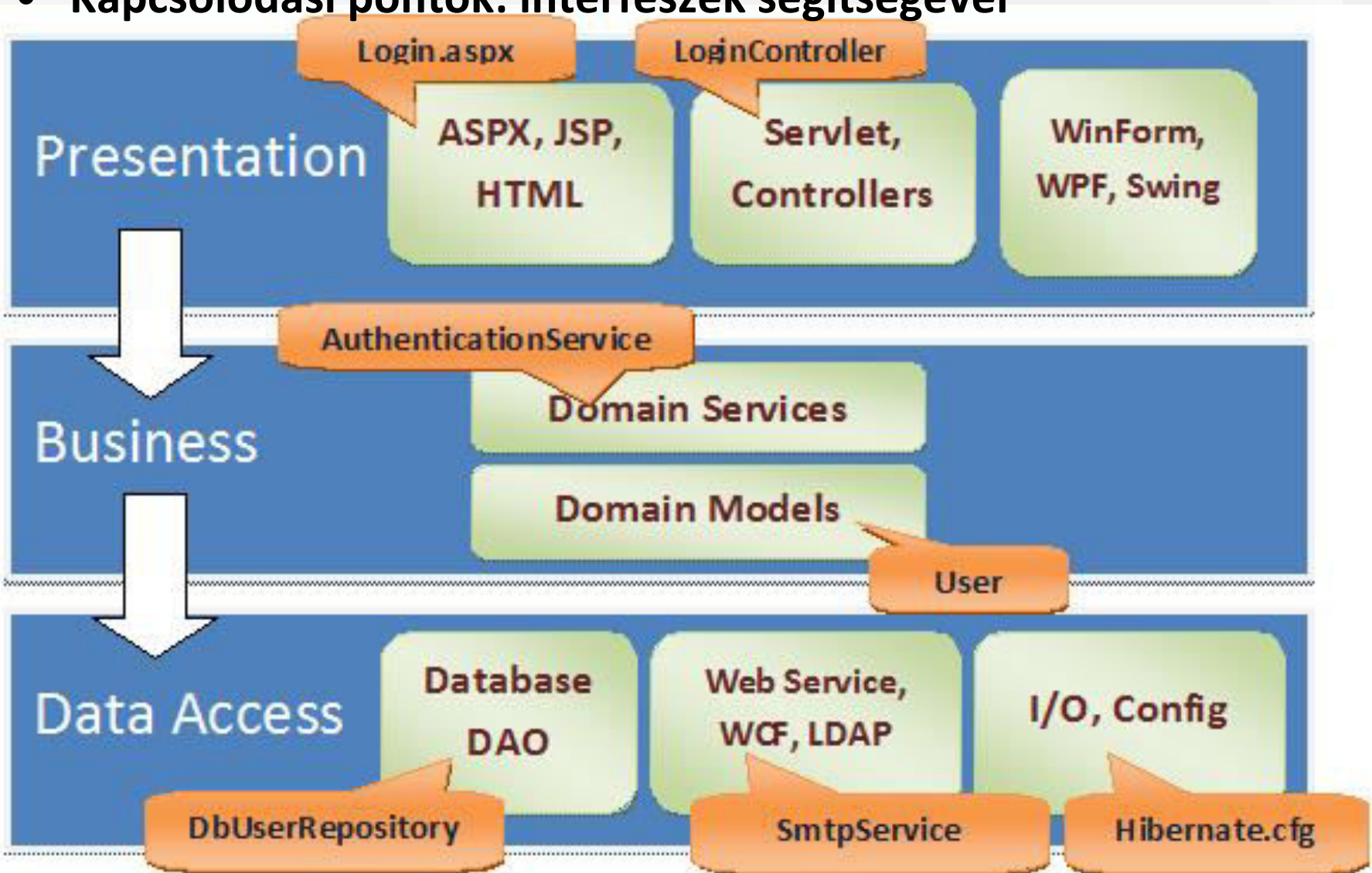
- **Tier: fizikai felbontása az alkalmazás komponenseinek**

- Szoftver- vagy hardver-komponensek, amik a rétegeket futtatják
- Nem feltétlenül azonos felbontású, mint a rétegek



Tipikus Szoftver rétegek

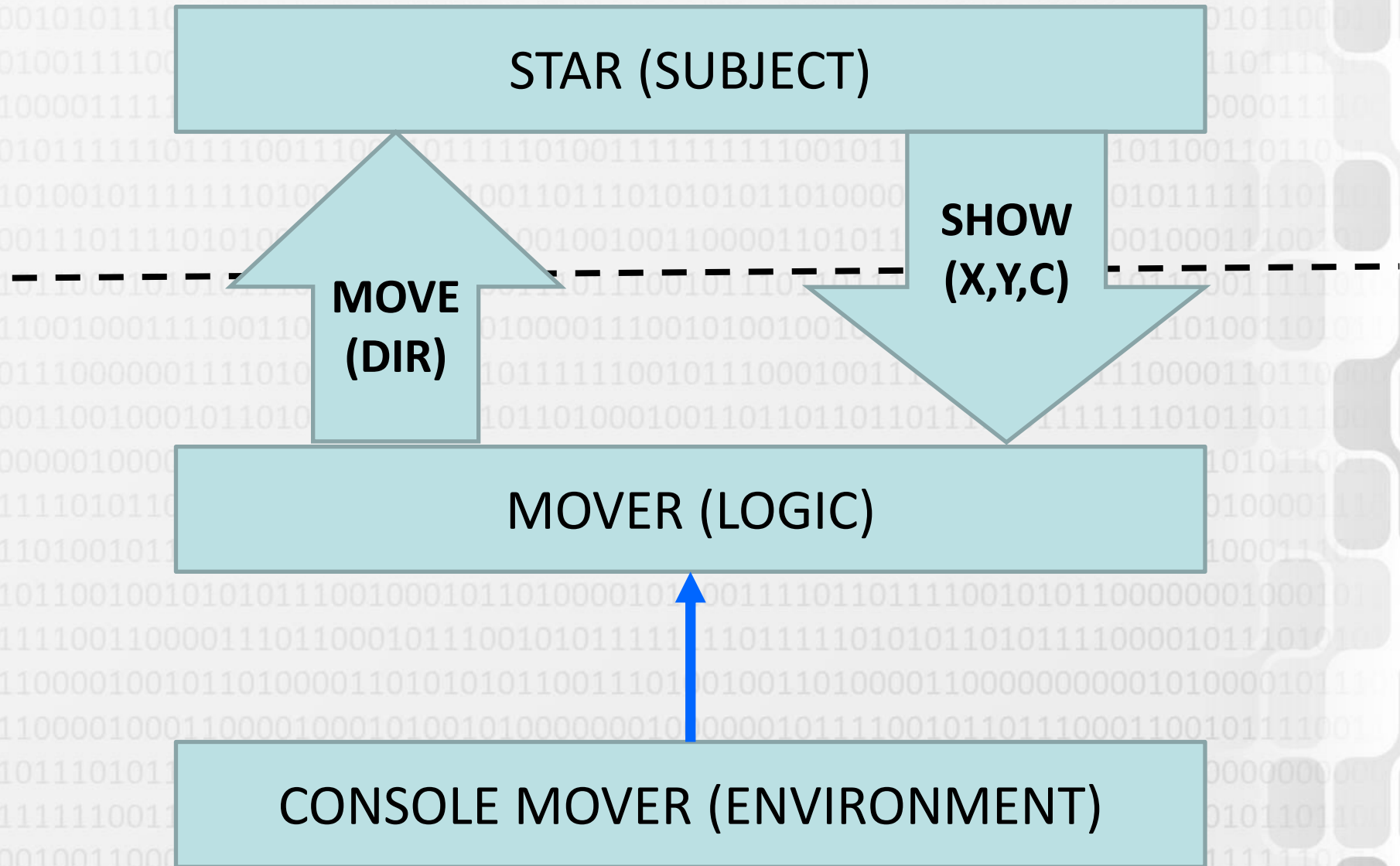
- Mindegyik réteg felbontható osztályokra/rétegekre
- Kapcsolódási pontok: interfészek segítségével



Nem rétegzett alkalmazás

```
int x = 0, y = 0;
while (true)
{
    ConsoleKeyInfo info = Console.ReadKey(); // CHANGE: different input
    switch (info.Key) // CHANGE: different logic
    {
        case ConsoleKey.UpArrow: y--; break; // TODO: range check
        case ConsoleKey.DownArrow: y++; break; // TODO: range check
        case ConsoleKey.LeftArrow: x--; break; // TODO: range check
        case ConsoleKey.RightArrow: x++; break; // TODO: range check
    }
    Console.Clear(); // CHANGE: different output
    Console.SetCursorPosition(x, y);
    Console.Write('*'); // CHANGE: different subject
}
```


Layered + Reuseable megvalósítás



SOLID elvek

- **S = Single Responsibility (*)**

- Egy jó osztály egy felelősségi körrel rendelkezik
- Nem szabad olyan osztályt készíteni, ami egyszerre adatkezelő, megjelenítő, műveletvégző, webszolgáltatás, ...

- **O = Open/Closed principle**

- Egy jó osztály egyszerre CLOSED (= használható) és OPEN (= bővíthető)
- Virtuális metódusok felüldefiniálásával, leszármazott osztályokkal elérhető

- **L = Liskov substitution**

- Ős helyébe tetszőleges utód példányosítható, ez nem ront a funkcionalitáson

- **I = Interface segregation**

- Sok kisebb interfész jobb, mint kevés nagyon nagy interfész

- **D = Dependency inversion (*)**

- Konkrét osztályoktól ne függjünk, helyette interfésztől / absztrakt osztálytól
- A függőség konkrét létrehozása VALAKI MÁS feladata...

- **Részletesebben: Prog4 (az csillaggal jelöltek a prog3 fő céljai)**

D = Dependency Inversion

- **Ne a konkrét X nevű osztálytól függjünk, hanem funkcionalitástól**
 - Interfész típus használatával VAGY absztrakt őssosztály használatával
- **Megkülönböztetendő: Dependency INVERSION vs Dependency INJECTION**
 - A dependency inversion elv többféle módon megvalósítható
 - Dependency injection (Prog3: interfész típusú ctor paraméterrel)
 - Factory design patterns (Prog4)
 - Inversion of Control (IoC) container/component segítségével (Prog4)
- **Ennek a félévnek elsődleges célja, hogy a SOLID elveknek megfelelő adatszerkesztő alkalmazást hozzunk létre, ami**
 - Rétegzett és adatforrás-független (ORM-et használ)
 - A logika réteg interfész típusú függőségeken keresztül, a Repository elvet követve éri el az adat-réteget
 - A logika réteg a dependency injection módszerrel kapja meg az adat réteget
 - A logika réteg tesztelhető: az adat réteg vagy fizikai adatbázis, vagy egy csak a tesztelést lehetővé tevő „kamu” adatforrás

MSSQL

- **MSSQL: tipikusan kis- és középvállalatok által használt relációs adatbázis-kezelő szerver**
 - SQL Express: kisebb változat: max 10GB/database, max 1 CPU, max 1GB RAM
 - VS (min. Community) telepítésekor „Data Storage and Processing”, vagy HDD hiány esetén SQL Server Data Tools + SQL Server 2016 Express LocalDB
 - LocalDb: Szerver-szolgáltatás helyett igény szerint induló library, ami egy adatbázis-file-t használ, nekünk ez kell!
- **VS Projekten belüli, in-solution „Service Based Database”**
 - LocalDB szolgáltatással megtámogatott MDF+LDF file-ok, a projekttel azonos könyvtárban (in-profile módszerrel NE)
 - Project / Add New Item / Service-based Database
- **Szervezés**
 - A GIT repónak is legyen része (.gitignore szerkesztése!!!)
 - Mindig legyen az EXE mellé bemásolva: MDF és LDF file esetén is jobb katt / Properties / Content + Copy Always
 - A táblák minden futtatáskor visszaállnak az eredeti állapotra!
 - SQL First vs Code First

ADO.NET: DbConnection/DbReader

- „Kapcsolt” adatbázis-elérés (Connected Data-Access Architecture)
- Előny: gyors, egyszerű
- Hátrány: nehéz módosítani és technológiát/tárolási módszert váltani; kapcsolat elveszését kezelni kell
- A különböző adatbázis-szerverekhez különböző implementációk
- Közös őszosztályok a különféle feladatokhoz
 - Adatbázis-kapcsolat: DbConnection
 - SQL/RPC utasítás végrehajtása: DbCommand
 - Utasítás eredményének beolvasása: DbDataReader
- Specifikus utódosztályok a különféle adatbázis-szerverekhez
 - SqlConnection (MSSQL System.Data.SqlClient)
 - MySqlConnection (MySQL MySql.Data.MySqlClient)
 - NpgsqlConnection (PostgreSQL - Npgsql)
 - OracleConnection (Oracle System.Data.OracleClient)

1. Inicializálás

```
string connStr = @"Data  
Source=(LocalDB)\v11.0;AttachDbFilename=path\to\empdept.mdf;  
Integrated Security=True;"
```

```
SqlConnection conn;
```

```
private void Connect()  
{  
    conn = new SqlConnection(connStr);  
    conn.Open();  
    Console.WriteLine("CONNECTED");  
}
```

2. INSERT

```
private void Insert()
```

```
{
```

```
    SqlCommand cmd = new SqlCommand("insert into EMP (ENAME,  
MGR, DEPTNO, EMPNO) values ('BELA', NULL, 20, 1000)", conn);
```

```
    int affected=cmd.ExecuteNonQuery();
```

```
    Console.WriteLine(affected.ToString());
```

```
}
```

3. UPDATE

```
private void Update()
```

```
{
```

```
    SqlCommand cmd = new SqlCommand("update EMP set  
    ENAME='JOZSI' where EMPNO=1000", conn);
```

```
    int affected=cmd.ExecuteNonQuery();
```

```
    Console.WriteLine(affected.ToString());
```

```
}
```


4. DELETE

```
private void Delete()
```

```
{
```

```
    SqlCommand cmd = new SqlCommand("delete from EMP where  
    EMPNO=1000", conn);
```

```
    int affected=cmd.ExecuteNonQuery();
```

```
    Console.WriteLine(affected.ToString());
```

```
}
```

5. SELECT

```
private void Select()
```

```
{
```

```
    SqlCommand cmd = new SqlCommand("select * from EMP where  
    SAL >= 3000 order by ENAME", conn);
```

```
    SqlDataReader reader = cmd.ExecuteReader();
```

```
    while (reader.Read())
```

```
    {
```

```
        Console.WriteLine(reader["ENAME"].ToString());
```

```
    }
```

```
    reader.Close();
```

```
}
```

5. SELECT

```
for (int i = 0; i < reader.FieldCount; i++) {  
    string coltext = reader.GetName(i).ToLower();  
    Console.WriteLine(coltext);  
}
```

```
for (int i = 0; i < reader.FieldCount; i++)  
{  
    Console.WriteLine(reader[i].ToString());  
    Console.WriteLine(reader.GetValue(i));  
    Console.WriteLine(reader.GetDecimal(i));  
}
```

A közvetlen SQL kommunikáció hátránya

- SQL Injection: Figyelni kell rá, hogy a usertől kapott adat SOHA ne értelmeződjön SQL utasításként

```
string uName = "yyy", uPass = "xxx";  
string sql = $"SELECT * FROM users WHERE  
username='{uName}' AND userpass=sha2('{uPass}');  
uPass = "x') OR 1=1 OR 1<>sha2('x";
```

- Az üzleti logikai kódba kell helyezni az SQL kódot
 - Nincs beépített védelem SQL Injection ellen
 - Módosítani kell SQL dialect vagy adatstruktúra módosítás esetén
 - Cél: az üzleti logika SOHA ne függjön az adattárolás módjától!
- Valamilyen réteg kell az SQL réteg fölé, ami elrejtí az SQL kódot
 - Prepared statements (sql command parameters)
 - Valamilyen erősen típusos, az SQL funkcionalitásával azonos (vagy ahhoz nagyon közelítő) réteget helyezünk az SQL réteg fölé
 - Dialektus-független lekérdezés = LINQ

DbConnection vs DataSet vs Entity Framework

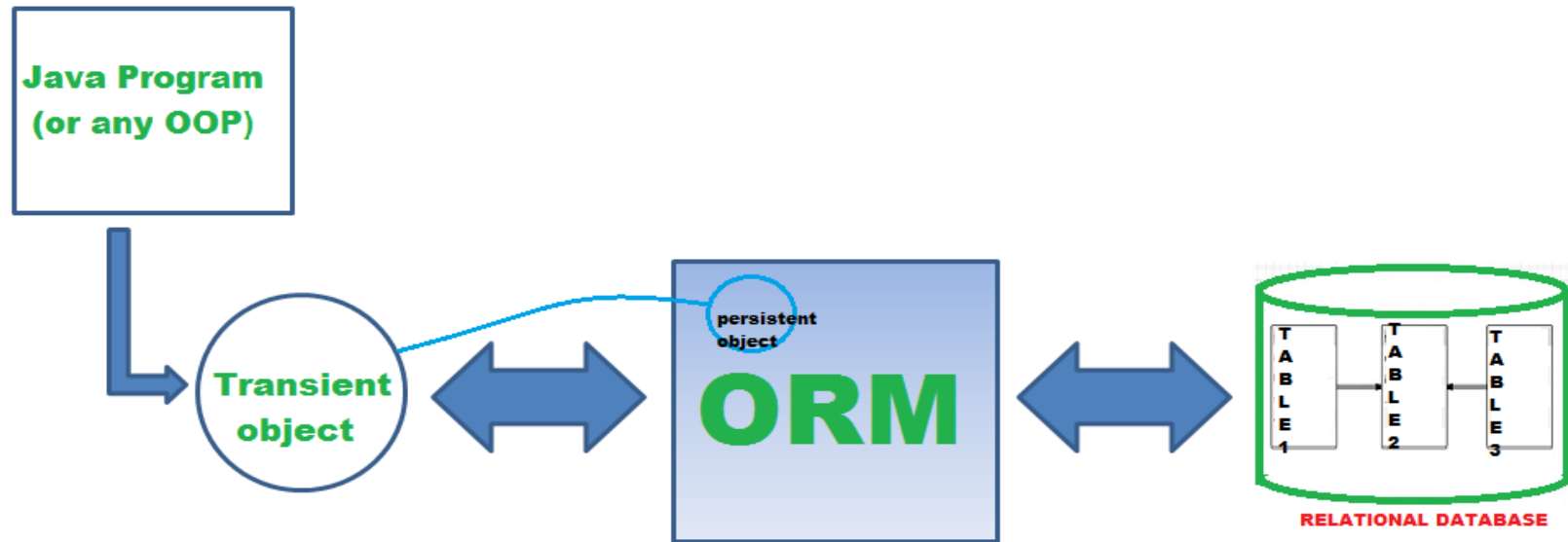
- **Cél: adatbázis-kapcsolat kezelése, SQL utasítások és eredmények feldolgozása**
- **DbConnection**
 - Alap SQL-szintű elérés, string SQL utasításokkal, object tömb eredményekkel
 - Connected mode
- **DataSet**
 - Az SQL réteg fölé egy erősen típusos, GUI központú réteg kerül, a műveleteket típusos metódusok végzik
 - Egyedi megközelítés, már nem használt
- **Entity Framework**
 - Az SQL réteg fölé ORM (Object Relational Mapping) réteget helyezünk: a táblákat mint objektumok általános gyűjteményét kezeljük
 - Tervezési mintákat felhasználó, általános megközelítés
 - Connected/Disconnected mode is

ORM = Object Relational Mapping – alapfeladat

- **REJTETT fizikai adatelérés, ami SQL utasításokkal dolgozik**
- **Kívülről használható műveletek: dialektus-függetlenség**
 - Dialektus-független konverzió: a műveletek végrehajtása független legyen attól, hogy milyen a használt SQL dialektus / adattárolási mód
 - A felső réteg ne SQL nyelvet lásson: Lambda kifejezések (Java Stream Api/C# LINQ) vagy saját lekérdező nyelv (Doctrine) vagy egyszerű CRUD metódusok (Active Record rendszerek)
- **Objektum-tár**
 - A felsőbb réteg csak típusos objektumokat lát
 - A lekérdezés eredmények objektumokká/gyűjteményekké konvertálódnak
 - Akár műveletek között is megoszthatóak
- **Az ORM segítségével a felsőbb réteg képes az adatbázist memóriában lévő objektumgyűjteményként kezelni, függetlenül a ténylegesen használt fizikai tárolási módtól**

ORM = Object Relational Mapping – rendszerek

- C#: Entity Framework
- Java: Hibernate/JPA
- Python: Django ORM, SQLAlchemy
- Ruby on Rails
- PHP: Eloquent, Propel, Doctrine



Doctrine DQL

```
$query = $this->em->createQuery( dql: "
    SELECT c
    FROM AppBundle:Choice c
    WHERE c.cho_visible=:visible
    ORDER BY c.cho_numvotes DESC")
->setParameter( key: 'visible', $isVisible);
```

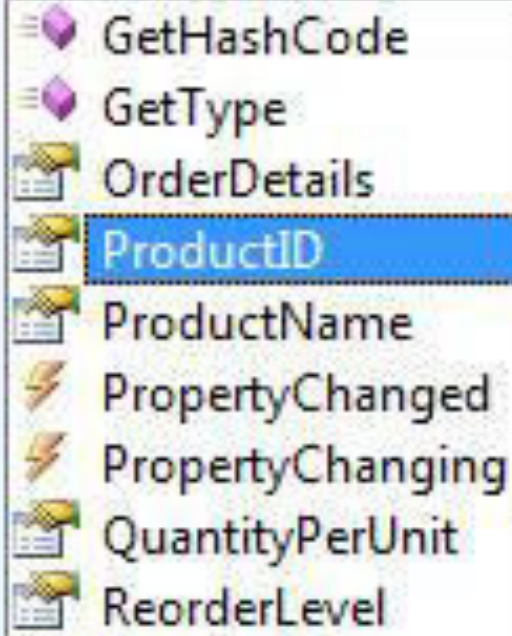
```
$qb = $this->em->createQueryBuilder();
$qb->select( select: 'c')
    ->from( from: 'AppBundle:Choice', alias: 'c')
    // ->innerJoin("c.cho_question", "q")
    ->where( predicates: 'c.cho_visible = :visible')
    ->orderBy( sort: 'c.cho_numvotes', order: 'DESC')
    ->setParameter( key: 'visible', $isVisible)
    ->getQuery();
return $query->getResult();
```

Java + Hibernate + Stream API

```
library.stream()  
    .map(book -> book.getAuthor())  
    .filter(author -> author.getAge() >= 50)  
    .map(Author::getSurname)  
    .map(String::toUpperCase)  
    .distinct()  
    .limit(15)  
    .collect(toList()));
```

C# + Entity Framework + LINQ

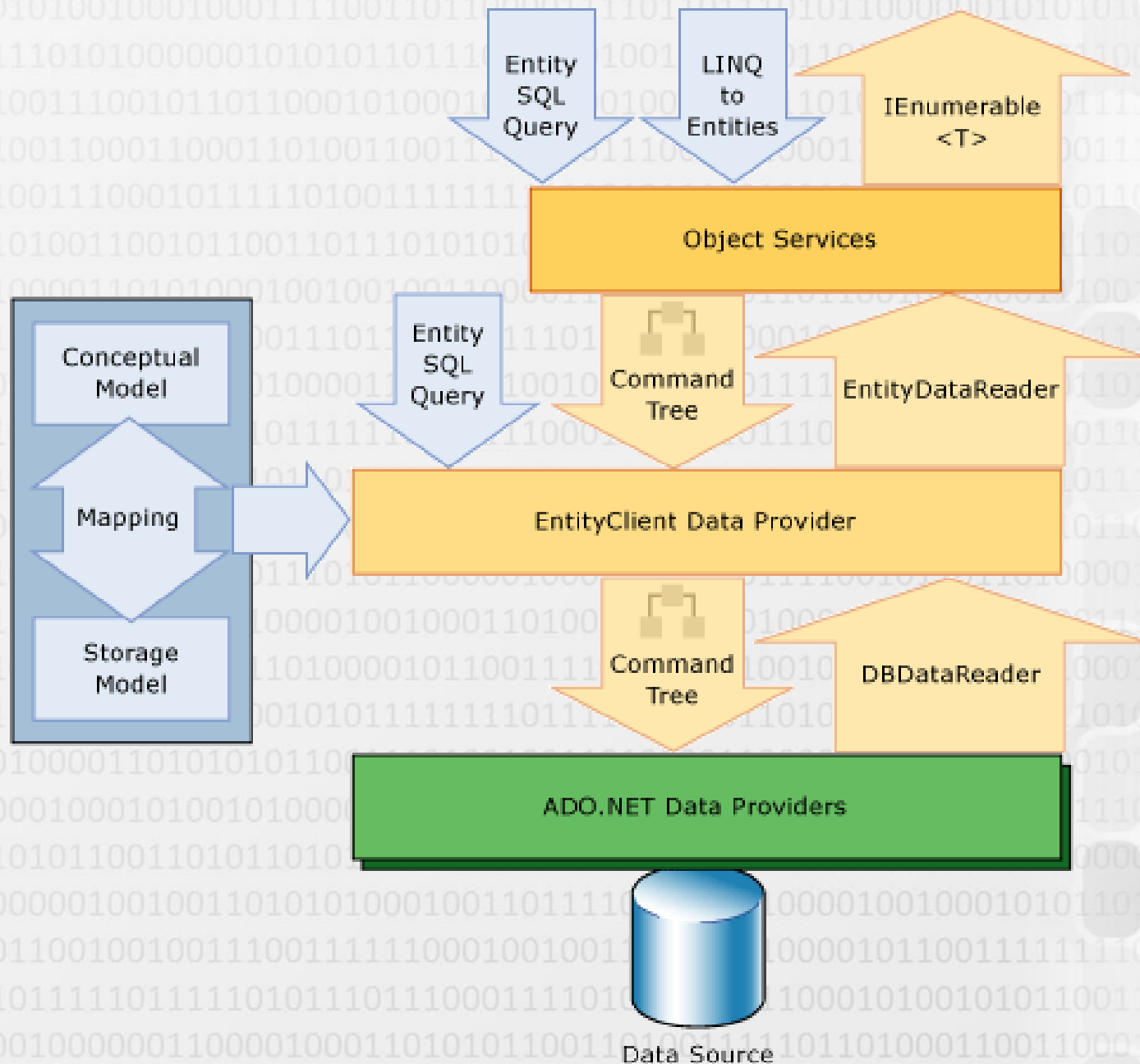
```
NorthwindDataContext db = new NorthwindDataContext();  
  
var products = from p in db.Products  
                where p.CategoryID == 2  
                select p;  
  
foreach (Product product in products)  
{  
    product.|  
}
```



- GetHashCode
- GetType
- OrderDetails
- ProductID**
- ProductName
- PropertyChanged
- PropertyChanging
- QuantityPerUnit
- ReorderLevel

int Product.ProductID

EF Rétegek



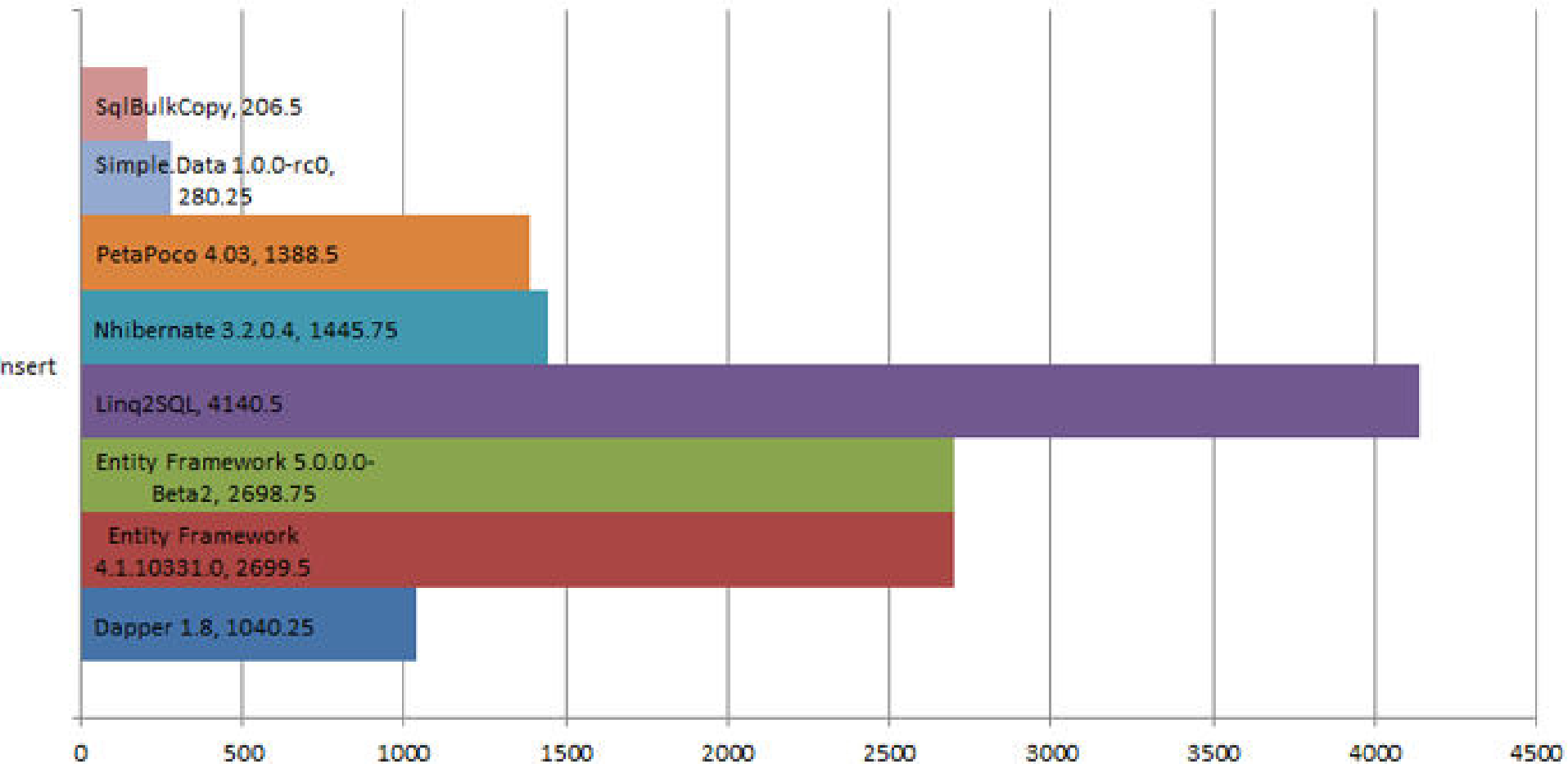
ORM használatának hátrányai

- **“ORM is a terrible anti-pattern that violates all principles of object-oriented programming, tearing objects apart and turning them into dumb and passive data bags. There is no excuse for ORM existence in any application”**

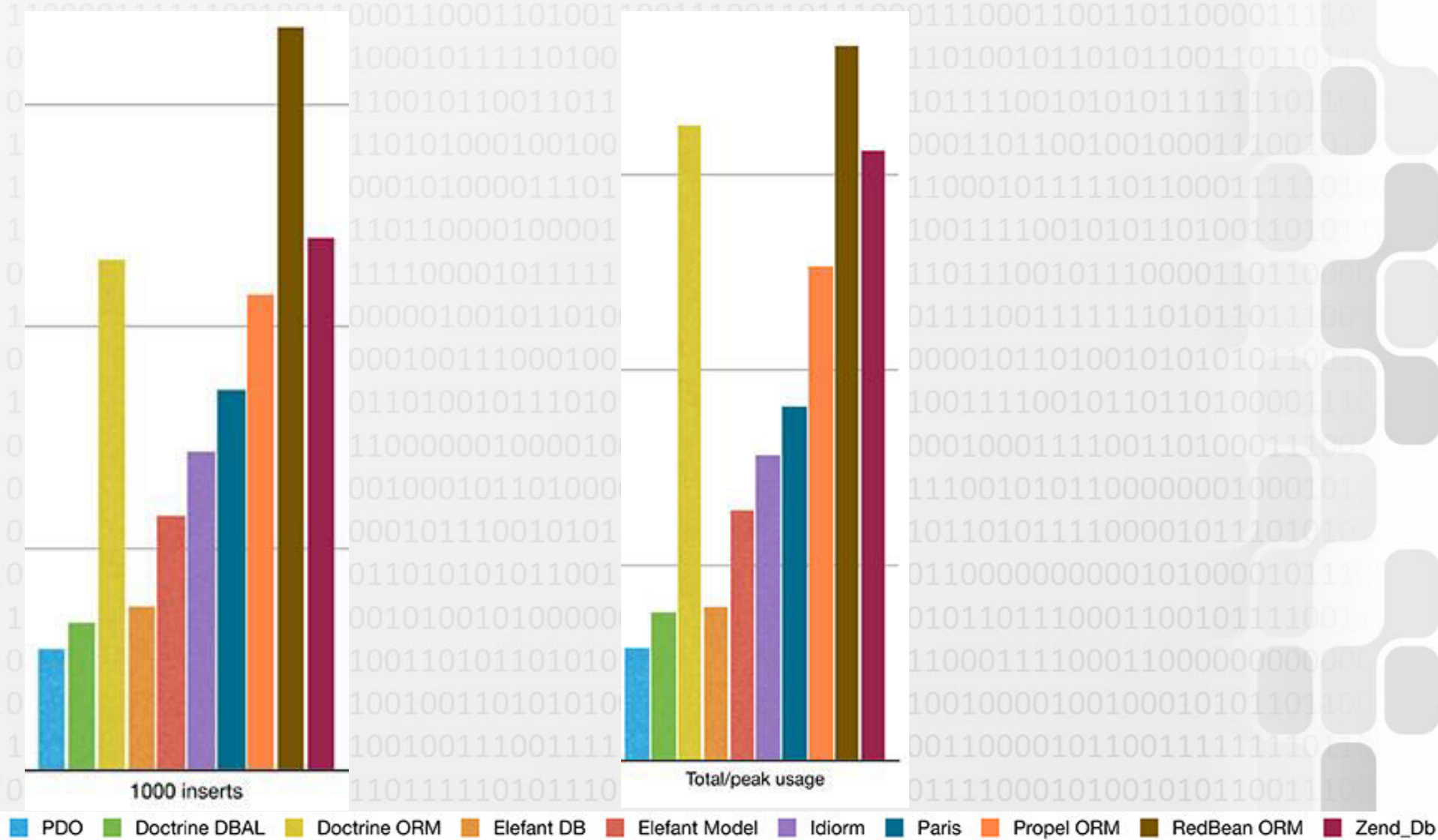
<http://www.yegor256.com/2014/12/01/orm-offensive-anti-pattern.html>

- **Ez persze csak egyetlen vélemény, de tipikus negatívumok:**
 - Nehezebb konfiguráció
 - Nagyobb memóriaigény
 - Nagyobb CPU igény
 - Bonyolultabb lekérdezések szegényes/nehéz támogatása
 - Nehéz optimalizáció

ORM sebesség (C#)



ORM sebesség (PHP)



ORM használatának előnyei

- **A kódban sehol sem használunk dialektus függő SQL utasításokat**
 - Bármikor dialektust/szerveret válthatunk a kód átírása nélkül
 - Saját EF/Linq provider írásával akár tetszőleges tárolási mód is lehetséges
- **A string formátumú SQL utasítások helyett a fordítás közben szintaktikailag ellenőrzött lekérdező formátumot használunk**
 - A fordítás közben kiderül, ha a bárhol szintaktikai hiba van
- **A string formátumú SQL paraméterek (string összefűzés) helyett változókat használunk lekérdezés paraméterként**
 - SQL injection elkerülése
- **A lekérdezések eredménye nem általános object / object[] / asszociatív tömb / típus nélküli Dictionary**
 - Helyette erősen típusos ismert típusú érték / példány / gyűjtemény (!!!)
- **Az ORM rétegre +1 réteg elhelyezésével könnyedén megoldható az adatforrás tetszőleges cseréje és a kód tesztelése**
 - Repository with Dependency Injection

+1 réteg ???

- **Előny, hogy ugyanaz a LINQ query fut**
 - Attól függetlenül, hogy milyen adatforrást használ (az adatforrás lehet List<T>, XML, MySQL/Oracle/MSSQL Adatbázis ...)
 - Könnyebb leválasztani az üzleti logikáról az adatfeldolgozást végző kódot
- **A cél: a LINQ query úgy hivatkozzon az adatforrásra, hogy az ne feltétlenül a fizikai adatbázist használja**
 - Cél: olyan Logic kódot írni, amelyben futásidőben változtatható, hogy az adatokat honnan szedje
 - Cél: olyan Logic kódot írni, amelyhez könnyű igazán jó egységteszteket írni (üzleti logika tesztelése ténylegesen működőképes, ténylegesen használt adatelérő réteg nélkül)
 - *(Talán megoldás, ebben a félévben biztos nem: EF Core In-Memory Providers)*
- **A Féléves Feladatban ez az elvárás!!!**
 - Tényleges Megoldás: Repository Design Pattern + Moq

Entity Framework verziók

<https://docs.microsoft.com/en-us/ef/ef6/what-is-new/past-releases>

- Linq To SQL: Formailag hasonló, „Félkész termék”
- EF1 = EF3.5 → .NET 3.5 (2008)
- EF4 → .NET 4 (2010) „POCO support, lazy loading, testability improvements, customizable code generation and the Model First workflow”
- EF4.1 → „first to be published on NuGet. This release included the simplified DbContext API and the Code First workflow” (2011) → a furcsaObjectContext api helyére a logikus DbContext api került!
- EF4.3 → „Code First Migrations” (2012) → használható ORM!
- EF5 (2012), API változások ...
- EF6 (2013), API változások ...
- EF 6.1 (2014), EF 6.2 (2017), EF 6.4 (2019) → elterjedt és JÓ ORM!

Entity Framework Core

- **EF7: RC1 2015 ...**
 - „You should view the move from EF6.x to EF Core as a port rather than an upgrade”
- **EF Core 1.0 (2016), 2.0 (2017), 2.1 (2018) (all with ASP.NET Core)**
 - ➔ **LAZY LOAD + GROUP BY, elértünk az EF 6.1 szintjére...**
- **EF Core 2.2 (2018), 3.0, 3.1 (2019)**
 - <https://docs.microsoft.com/en-us/ef/core/what-is-new/ef-core-3.x/breaking-changes>
 - <https://www.infoq.com/news/2019/06/EF-Core-3-Breaking-Changes/>
- **Jelenleg:**
 - EF 6.4 ➔ régi kódbázis**
 - EF Core 3.1 ➔ Útban a .NET Framework 5 felé**
- **<https://docs.microsoft.com/en-us/ef/efcore-and-ef6/>**
 - Többnyire minden funkció elérhető, néhány kisebb hiányosság, néhány jó újítás
 - Limited inheritance models (planned for 5.0), No change Tracking Proxies (merged in 5.0), limited many-to-many (planned for 5.0), No database.log (merged in 5.0)
 - In the backlog: Update model from database, Stored procedures

DbContext API

- Minden táblához szükség van egy, a táblával „azonos struktúrájú” osztályra: **Entity class / Entity model**
 - A „Data Mapper” lényege, hogy elvileg lehet más az osztály és a tábla
 - N:M kapcsolatnál az „entityless” mód: kapcsolótábla entity nélkül
 - Ebben a félévben ezeket ne próbáljuk ki:
1 tábla = 1 entity class, SQL adatmezők = C# property-k
- **DbContext (Model / Entity Model)**
 - Magát az adatbázist reprezentáló osztály
 - Feladata a Connection kezelése és a táblák egységbe zárása
 - A saját adatbázist ennek leszármazott osztálya fogja kezelni
 - OnConfiguring() metódusa a kapcsolat beállítását végzi
 - OnModelCreating() metódusa a táblák/entitások beállítását végzi:

FLUENT API
- **DbSet<T> : IQueryable<T>**
 - Táblát (= T típusú objektum gyűjteményt) reprezentáló osztály
 - A DbContext-ben 1-1 property minden táblának

Fluent API vs Annotations

- **Közös cél: a kódban definiálni, hogy az entity osztályokban lévő C# tulajdonságok specifikus extr**

```
[Table("cars")]  
public class Car  
{
```

- **Attribútummal attribútumokk**

```
    [Key]  
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]  
    [Column("car_id", TypeName = "int")]  
    public int Id { get; set; }
```

– Cars + Brands

- **Fluent API: DbContext leszármazott metódusában**

- Modernebb, jobban illeszkedik az EF
- EmpDept példa
- Idegen kulcsokhoz SOKKAL JOBB (Wit

```
entity.HasKey(car => car.Id)  
    .HasName("CAR_PRIMARY_KEY");  
  
entity.Property(car => car.Id)  
    .UseIdentityColumn() // IDENTITY  
    .HasColumnName("car_id")  
    .HasColumnType("int");  
  
entity.ToTable("cars");
```

- **A féléves feladatban mindegy, melyiket használjuk**

- Annotációs módszer esetén is biztosan kell az OnModelCreating()
- Idegen kulcsok és Database Seeding miatt

A modell legenerálása (RÉGI SQL first)

- **Előfeltétel: EF 6.x és meglévő táblák**
<http://msdn.microsoft.com/en-us/data/jj206878>
- **Project, Add new Item, ADO.NET Entity Data Model <ADD>**
 - Generate from database <NEXT>
 - Az MDF file legyen a legördülő menüben + save connection settings <NEXT>;
 - EF6.0 <NEXT>;
 - Mindegyik tábla mellett pipa + Model namespace = EmpDeptModel <FINISH>
- **Konfigurációtól függően: Template can harm your computer, click ok to run ... <OK> <OK>**
- **Eredmény: automatikusan generált erősen típusos osztályok, csak ezek vagy generikus osztályok típusparaméterezett változatai vagy inkább POCO osztályok ➔ ~30KB a két táblás adatbázis (DataSet: 10x ekkora)**

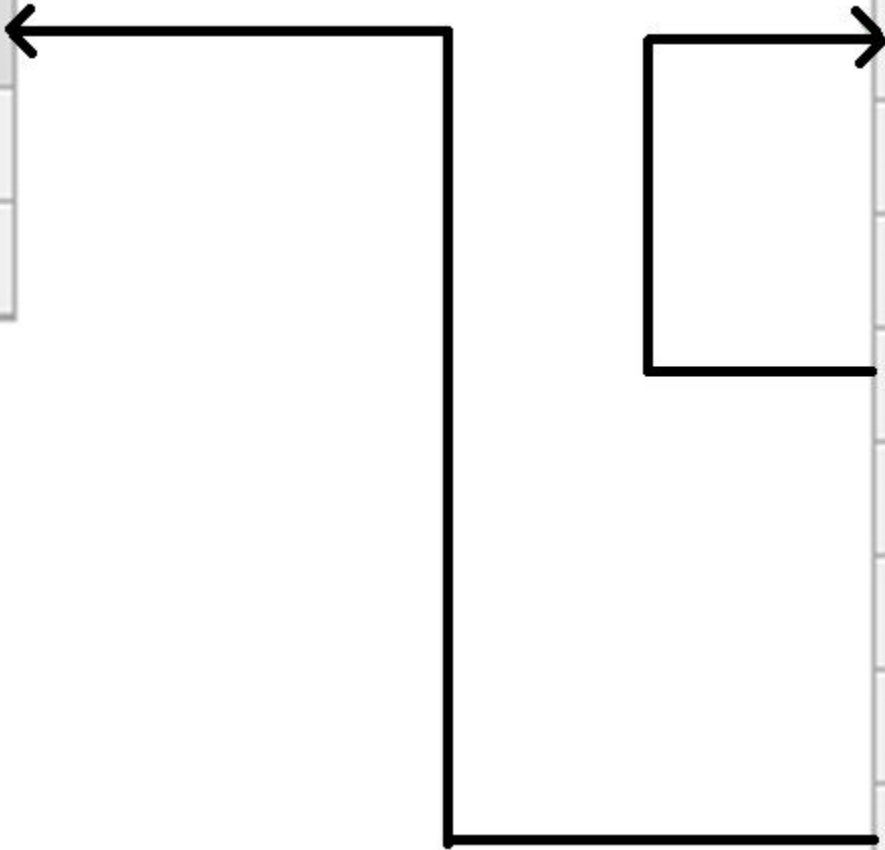
A modell legenerálása (ÚJ EF Core SQL first)

- **EmpDept.mdf; meglévő táblák; Content+Copy Always**
- **Server Explorer → EmpDept → Right click → Properties**
 - "Data Source = (LocalDB)\MSSQLLocalDB; AttachDbFilename = |DataDirectory|\EmpDept.mdf; Integrated Security = True"
 - „Close Connection” fontos a Server Explorer-ben!!!
- **Manage Nuget Packages for Solution => NOT prerelease**
 - Microsoft.EntityFrameworkCore.SqlServer
 - Microsoft.EntityFrameworkCore.Tools
 - Microsoft.EntityFrameworkCore.Proxies
- **Nuget Package Manager Console**
 - Scaffold-DbContext "CONNECTION_STRING" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
- **Code first megközelítésnél teljesen azonos szintaxisú osztályokat kell kézzel megírni, mint amit a Scaffold-DbContext legenerál**
 - Szükséges az üres MDF+LDF állomány, valamint a connection string

Példa Táblák

| | | | |
|--------------------------|--------|-----|--|
| | DEPT | | |
| <input type="checkbox"/> | DEPTNO | 789 | |
| <input type="checkbox"/> | DNAME | A | |
| <input type="checkbox"/> | LOC | A | |

| | | | |
|--------------------------|----------|-----|----|
| | EMP | | |
| <input type="checkbox"/> | EMPNO | 789 | |
| <input type="checkbox"/> | ENAME | A | |
| <input type="checkbox"/> | JOB | A | |
| <input type="checkbox"/> | MGR | 789 | |
| <input type="checkbox"/> | HIREDATE | | 31 |
| <input type="checkbox"/> | SAL | 789 | |
| <input type="checkbox"/> | COMM | 789 | |
| <input type="checkbox"/> | DEPTNO | 789 | |



1. Inicializálás

```
ED = new EmpDeptEntities();
```

```
Console.WriteLine("Connect OK");
```

```
// DbSet<T> típus, LINQ bővítménymetódusokkal
```

```
// LINQ query syntax is lehetséges
```

```
// IEnumerable<T> vs IQueryable<T>
```

```
Dept dept = ED.Dept.First();
```

```
Console.WriteLine(dept.Dname);
```

```
// Lazy loading
```

```
string deptName = ED.Emp.First().Dept.Dname;
```

2. INSERT

```
Emp newWorker = new Emp()
```

```
{
```

```
    Ename = "BELA",
```

```
    Mgr = null,
```

```
    Deptno = 20,
```

```
    Empno = 1000
```

```
};
```

```
ED.Emp.Add(newWorker);
```

```
ED.SaveChanges(); // Save via change tracking!
```

```
Console.WriteLine("Insert OK");
```

3. UPDATE

```
Emp someone = ED.Emp.Single(x => x.Empno == 1000);  
someone.ENAME = "JOZSI"; // Change tracking!  
ED.SaveChanges();  
Console.WriteLine("Update OK");
```

4. DELETE

```
Emp someone = ED.Emp.Single(x => x.Empno == 1000);
```

```
ED.Emp.Remove(someone);
```

```
ED.SaveChanges();
```

```
Console.WriteLine("Delete OK");
```


5. SELECT / Eager vs Lazy Loading

- **Eager Loading: Jobb Teljesítmény, DE előre tudni kell a később elérni kívánt hivatkozásokat, és ezeket a lekéréskor be kell hivatkozni**

```
foreach (var item in ED.Emp.Include(emp => emp.DeptnoNavigation))
{
    Console.WriteLine($"{item.Ename} in {item.DeptnoNavigation.Loc}");
}
```

- **Lazy Loading: Gyengébb teljesítmény (főleg SOK rekordnál), de SOKKAL kényelmesebb használni, mert nem kell semmit előre behivatkozni az adat lekérésekor**

```
foreach (var item in ED.Emp)
{
    Console.WriteLine($"{item.Ename} in {item.DeptnoNavigation.Loc}");
}
```

- UseLazyLoadingProxies() kell az OnConfiguring() –ba, ez után vagy a .ToList() – en kell végigmenni, vagy pedig a „MultipleActiveResultSets = true” kell a connection stringbe

Példa: az EmpDept adatbázison...

1. **Hozza létre az adatbázist, majd abból SQL First módszerrel a modellt**
Nézze át a generált Entity és DbContext osztályokat
2. **Listázza a dolgozók neveit a részlegük helyével együtt**
3. **Listázza a dolgozók nevét, fizetését, munkakörét, valamint a munkakör átlagjövedelmét (jövedelem = Sal + Comm)**
4. **Listázza a dolgozók nevét, részlegük nevét, valamint a részlegükben dolgozó személyek darabszámát**
5. **Duplázza meg az elnök (PRESIDENT) fizetését**
6. **Törölje azokat, akik az elnök belépése után kevesebb, mint 30 nappal léptek be a céghez**

Gyakorlat: Cars + Brands adatbázis létrehozása

1. Hozza létre az adatbázist, majd abba teljesen Code First módszerrel töltse bele a táblákat és az adatokat (annotációkkal is megoldható akár – de a Fluent API teljes használata is ugyanúgy jó)
2. Listázza az összes márkát
3. Listázza az összes autót, a márkánévvel együtt (Lazy Load módszerrel)
4. Listázza az autókat, a márkák átlagárával együtt

