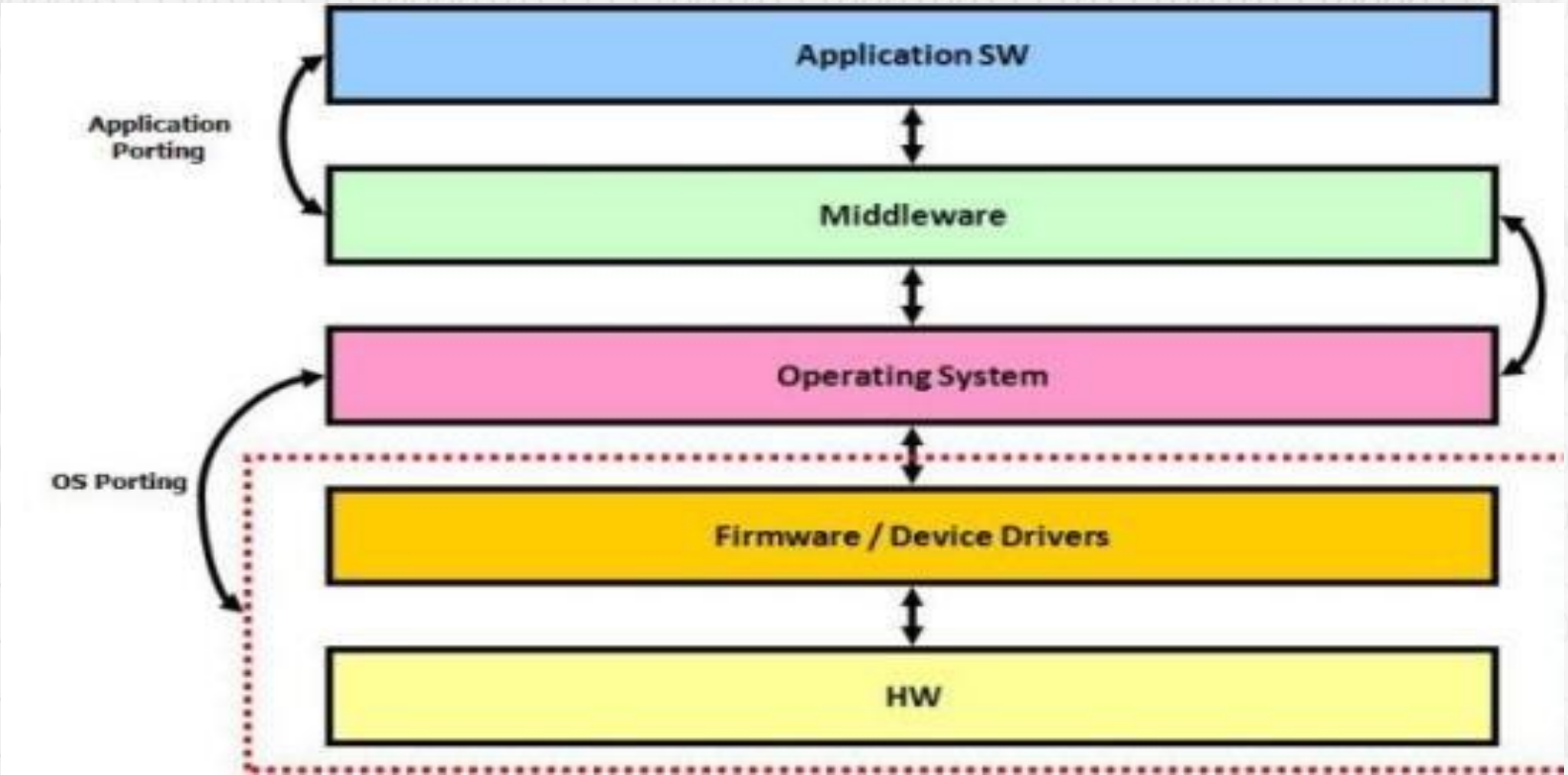
Advanced Programming Techniques

Layering, SOLID principles
Accessing databases using the DbConnection/DbReader method
Accessing LocalDb using Entity Framework Core (ORM + LINQ)

Layering, SOLID principles

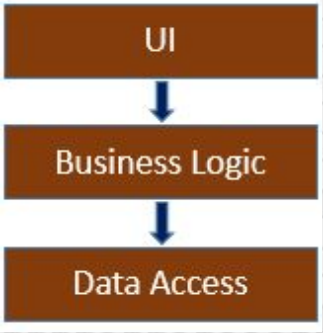
Layering

- **Aim: hide the implementation details so that the “upper” layer does not depend on the “lower” layer that is far away**
- **Same principle was already used many-many areas**
  - OS/Network, different abstraction layers
- **Every layer is depending <u>only</u> on their bottom neighbour**



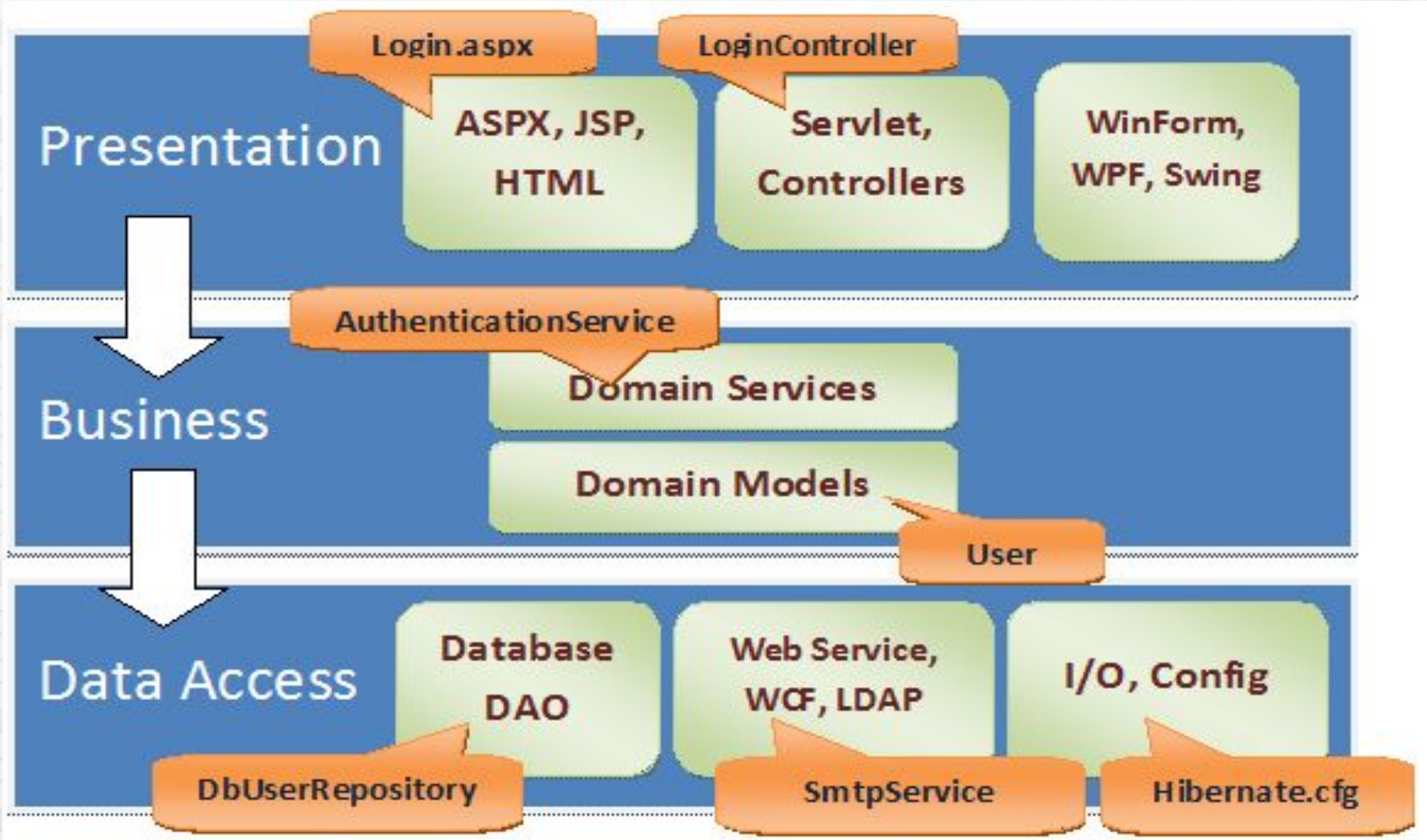
Tier vs Layer

- **Layer: logical separation of classes**
  - The classes belonging to the same layer serve a “higher purpose” together (e.g. user interface, data management, logic…)
  - It is clearly defined which classes/operations are accessible externally from other layers
  - The external (used and provided) functionalities are described using interfaces
  - We have a well written software if a layer (as a whole) can be fully replaced with a different one that works with the same interfaces
- **Tier: physical separation of the application components**
  - Software or hardware components that execute layers
  - Not necessarily the same structure as the layers



Typical Software Layers

- All can be split up into sub-layers / classes
- Connecting points: using interfaces
- Every layer only communicates to the bottom one

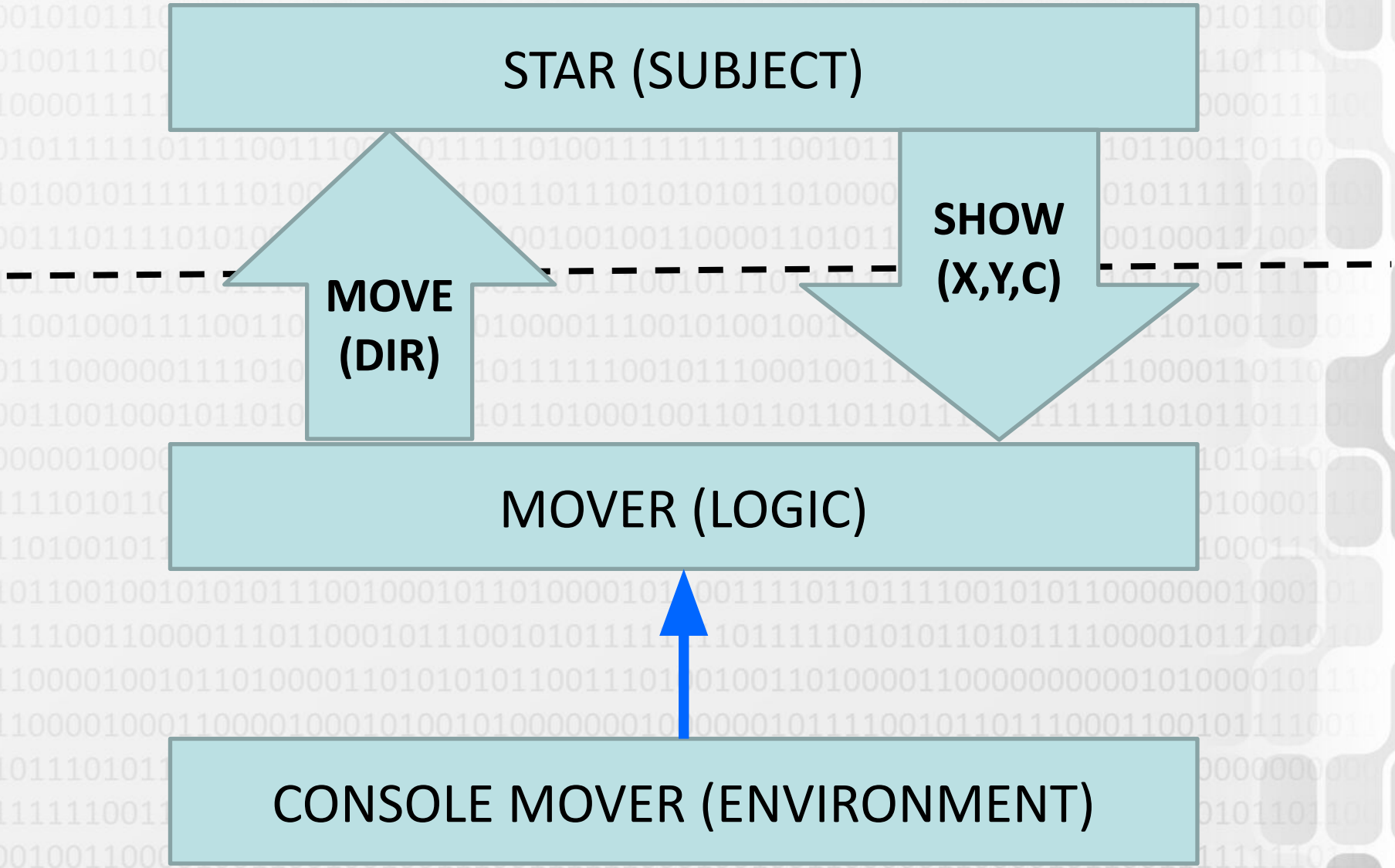


Non-Layered application

Problem: Everything is packed together, no “logic” and no “ui” separation.

```
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 solution



Reaching layered application

In order to reach a layered design, our code (classes, methods, interfaes etc.) MUST follow some rules.

We need layered design because

- **code will be more clear (vs write everything into 1 file)**
- **better to understand**
- **easier to maintain**
- **easier to extend and add new features**

These rules will be the SOLID principles, which are one of the most important part of this semester.

SOLID principles

- **S = Single Responsibility (*)**
  - A good class only has a single area of responsibility
  - We must not create a class that does data processing, user display, executes operations, hosts a web service, …
- **O = Open/Closed principle**
  - A good class is simultaneously CLOSED (= useable) and OPEN (= extensible)
  - By overriding virtual methods and using descendent classes
- **L = Liskov substitution**
  - Any descendant instance must be useable in place of a base instance
- **I = Interface segregation**
  - Multiple smaller interfaces are better than a few large ones
- **D = Dependency inversion (*)**
  - Should not depend on concrete classes, rather on interfaces/abstract classes
  - Creation of concrete dependencies should be done by SOMEONE ELSE...

More details on next semester's STGUI

Marked ones are the main aims for ADT

SOLID principles

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

We already seen:
Creating a LinkedList class has the responsibility ONLY to store T items in it (and additional features like insert, get given index etc.)

- **O = Open/Closed principle**

We already seen:
Using descendant classes, with right visibility settings, using virtual methods etc.

- **L = Liskov substitution**

We already seen: Descendant object can be referenced with the base class. Person » Student
Person[] items = new Person[10] … items[0] = new Student(){...}

- **I = Interface segregation**

We already seen: IComparable » CompareTo » All that for only 1 method?! YES.

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

New stuff.

More details on next semester's STGUI
Marked ones are the main aims for ADT

D = Dependency Inversion

- **Should not depend on the concrete class, but on the functionality**
  - Using an interface type OR using an abstract base class reference
- **Not the same: Dependency INVERSION vs Dependency INJECTION**
  - The dependency inversion principle has multiple implementations
  - Dependency injection (ADT: using interface-typed ctor parameters)
  - Factory design patterns (STGUI)
  - Using an Inversion of Control (IoC) container/component (STGUI)
- **The main aim of this semester is to create a data processing app that complies with the SOLID principles**
  - Layered and datasouce-independent (uses an ORM)
  - The logic layer uses the Repository Pattern and interface-typed dependencies to access the data-layer
  - The logic layer receives the data layer using dependency injection
  - The logic layer is testable: the data layer can be a physical database or only a “fake” data source useable for testing purposes

Accessing databases using the DbConnection/DbReader method

MSSQL

- **MSSQL: a relational database management server used by smaller and medium-sizedlevel companies**
  - SQL Express: smaller & free: max 10GB/database, max 1 CPU, max 1GB RAM
  - When installing VS (min. Community) „Data Storage and Processing”, or if lacking HDD space SQL Server Data Tools + SQL Server 2016 Express LocalDB
  - LocalDb: Instead of a service, an on-demand loaded library that uses a database file – this is what we will use this semester!
- **Inside a VS project, in-solution „Service Based Database”**
  - LocalDB supported MDF+LDF files, inside the project folder (do NOT use the in-profile method)
  - Project / Add New Item / Service-based Database
- **Organization**
  - Must be part of the GIT repo (edit the .gitignore!!!)
  - Must be copied next to the EXE file: both the MDF and LDF files should have right click / Properties / Content + Copy Always
  - The tables are reset to originals at every execution!
  - SQL First vs Code First

ADO.NET: DbConnection/DbReader

- **Connected Data-Access Architecture**
- **Advantage: fast, simple**
- **Disadvantage: hard to modify and to change technology/storage method; must handle the loss of tcp connection**
- **Different implementations for different database servers**
- **Common base classes for various tasks**
  - Database connection: DbConnection
  - Execution of SQL/RPC statements: DbCommand
  - Read results of Select statements: DbDataReader
- **Specific descendent classes for the various servers**
  - SqlConnection (MSSQL System.Data.SqlClient)
  - MySqlConnection (MySQL MySql.Data.MySqlClient)
  - NpgsqlConnection (PostgreSQL - Npgsql)
  - OracleConnection (Oracle System.Data.OracleClient)

1. Initialization

```
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));
}
```

Disadvantages of direct SQL communication

- **SQL Injection: Must be careful NEVER to interpret any user input as part of the SQL statement**

```
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";
```

- **We must put the SQL code mixed up with the logic code**
  - No built-in protection against SQL Injection
  - We must modify it when changing SQL dialect or data structure
  - Aim: the business logic must NEVER depend on the physical data storage!
- **We must put a layer above the SQL layer that hides the SQL code**
  - Prepared statements (sql command parameters)
  - We need a strongly typed layer that supports the same (or similar) functionalities as the SQL language
  - Dialect-independent queries = LINQ

Accessing LocalDb using Entity Framework Core (ORM + LINQ)

DbConnection vs DataSet vs Entity Framework

- **Aim: handle database connections, process SQL statements and results**

better approach

DbConnection

- Basic SQL access with string SQL statements and object-array results
- Connected mode

DataSet

- A strongly typed, UI-centered layer above the SQL layer
- Operations are done using typed methods
- Unique approach, not used any more

Entity Framework

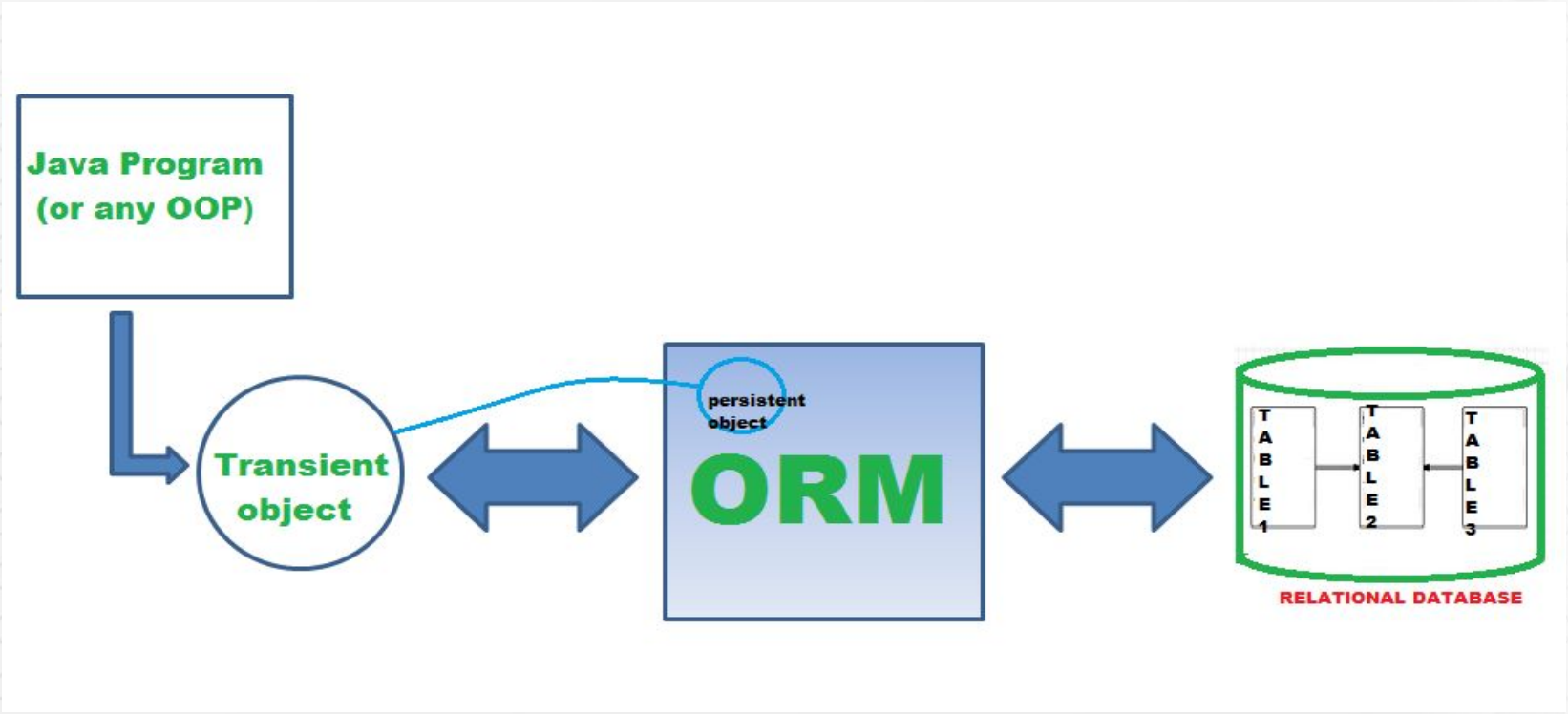
- A strongly typed ORM (Object Relational Mapping) layer above the SQL layer: we treat tables as generic storages for typed object instance
- General design approach that uses multiple design patterns
- Connected/Disconnected mode

ORM = Object Relational Mapping – task

- **HIDDEN physical data access that works with SQL statements**
- **Externally accessible operations: dialect-independent**
  - Dialect-independent conversion: the execution of the operations must be independent from the used SQL dialect / storage methods
  - The upper layer should not see the SQL language: Lambda expressions (Java Stream Api/C# LINQ) or a self-made query language (Doctrine) or simple CRUD methods (Active Record systems)
- **Object-storage**
  - The upper layer only sees typed objects
  - The query results are converted to objects/collections
  - Which can be shared between operations
- **Using the ORM approach, the upper layer can handle the database as an in-memory object storage, independent from the actual physical storage**

ORM = Object Relational Mapping – systems

- 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

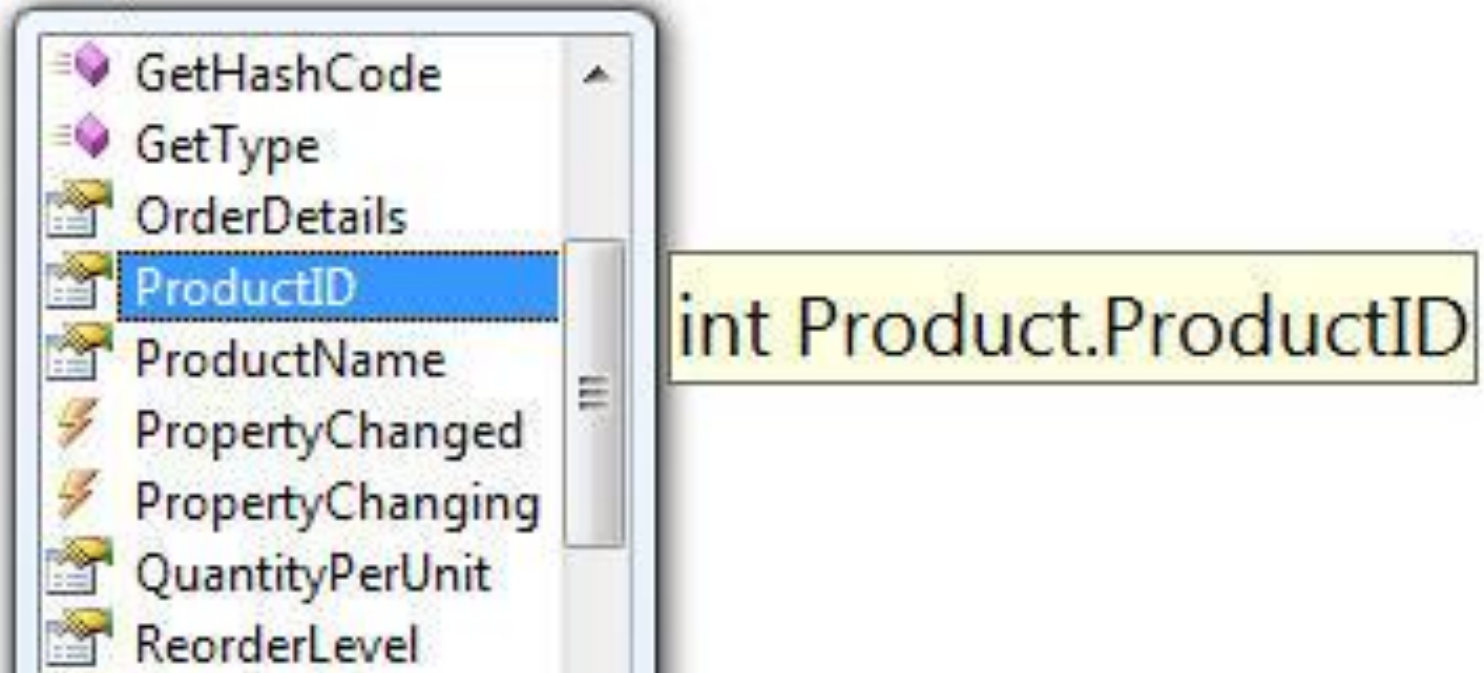
```
library.stream()
        .map(book -> book.getAuthor())
        .filter(author -> author.getAge() >= 50)
        .map(Author::getSurname)
        .map(String::toUpperCase)
        .distinct()
        .limit(15)
        .collect(toList()));
```

C# + Entity Framework + LINQ

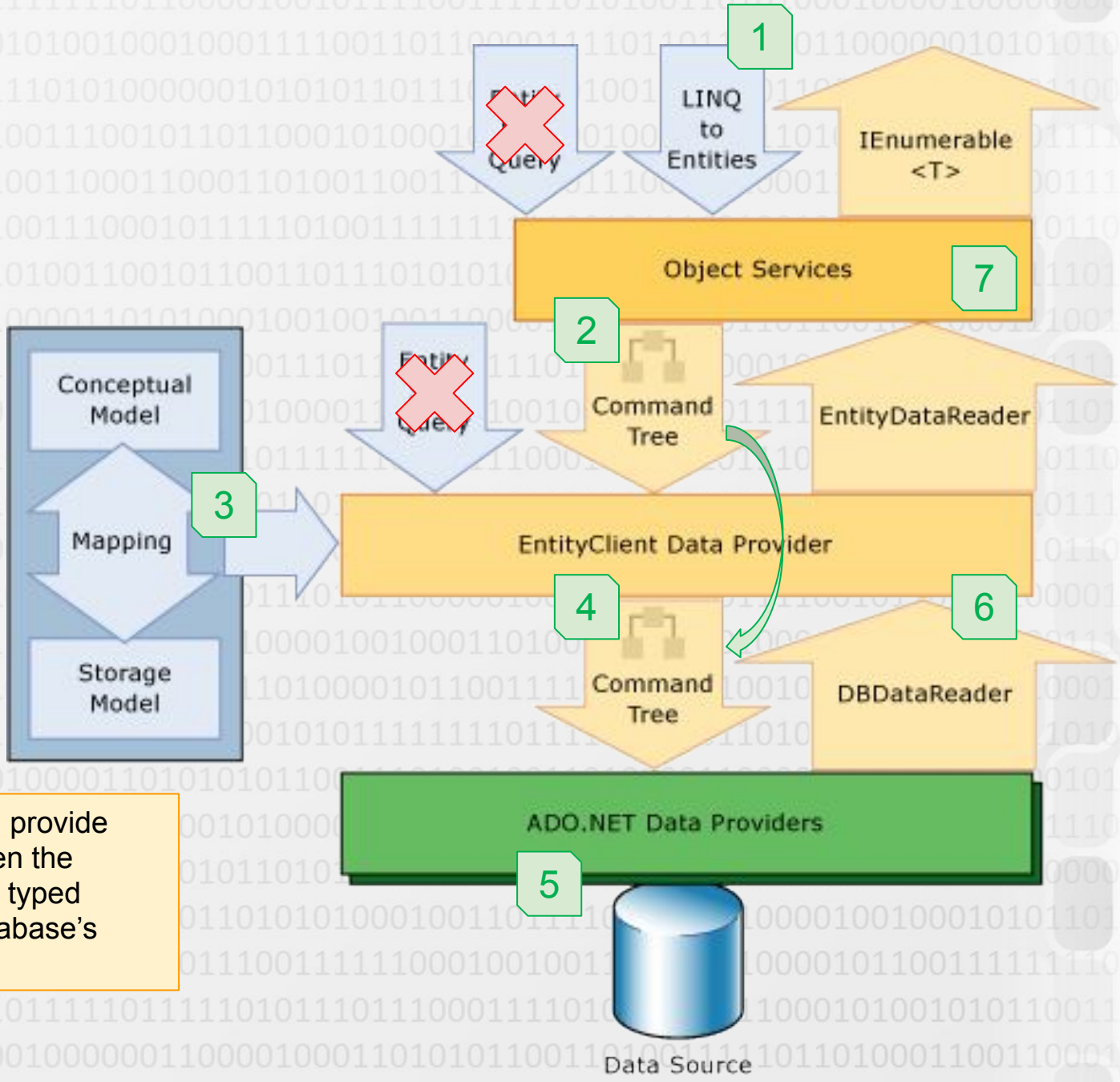
```
NorthwindDataContext db = new NorthwindDataContext();

var products = from p in db.Products
               where p.CategoryID == 2
               select p;

foreach (Product product in products)
{
    product.
}
```



EF layers



Mapper: create and provide **connection** between the language's strongly typed objects and the database's tables.

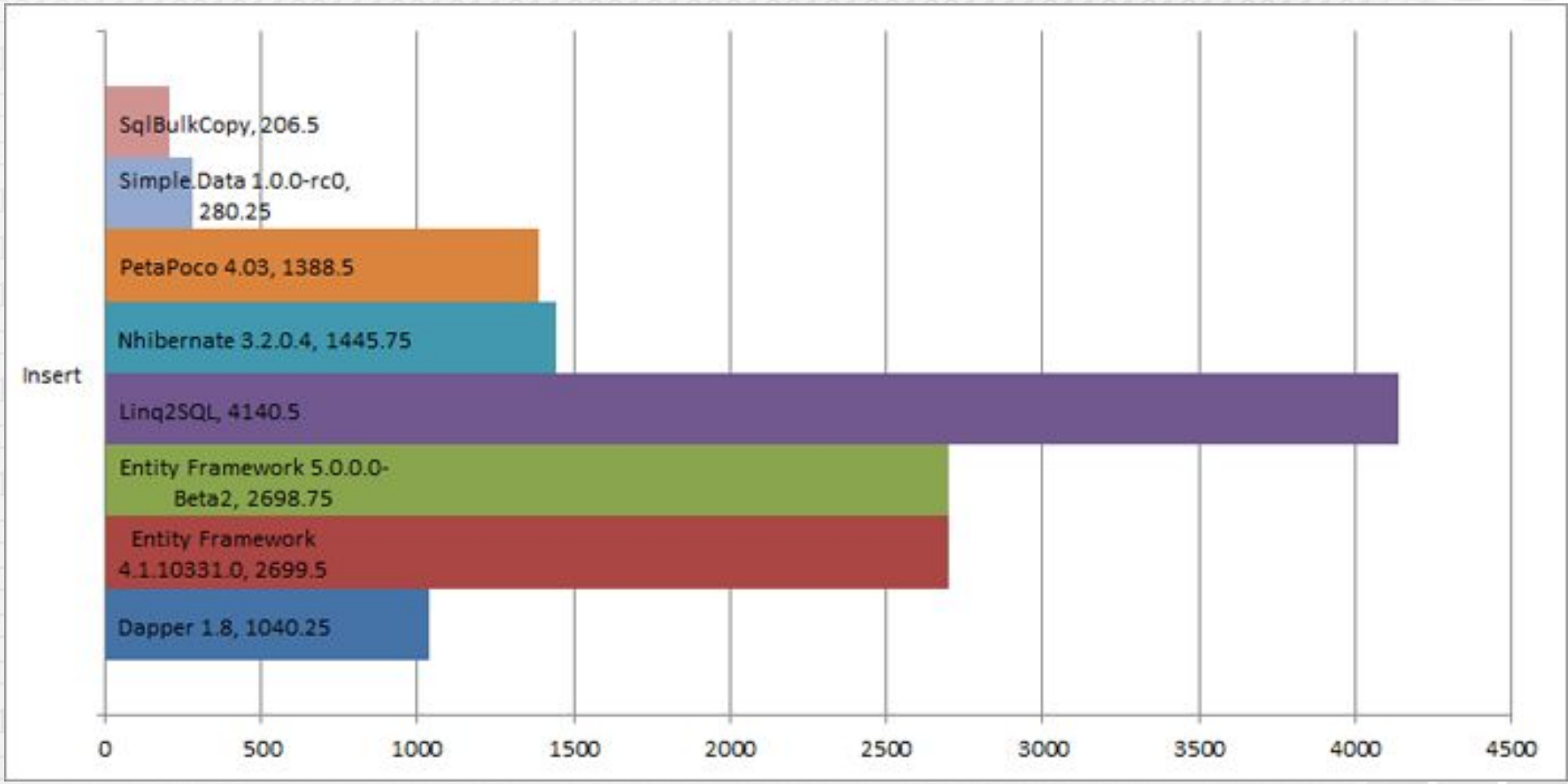
ORM disadvantages

- **“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

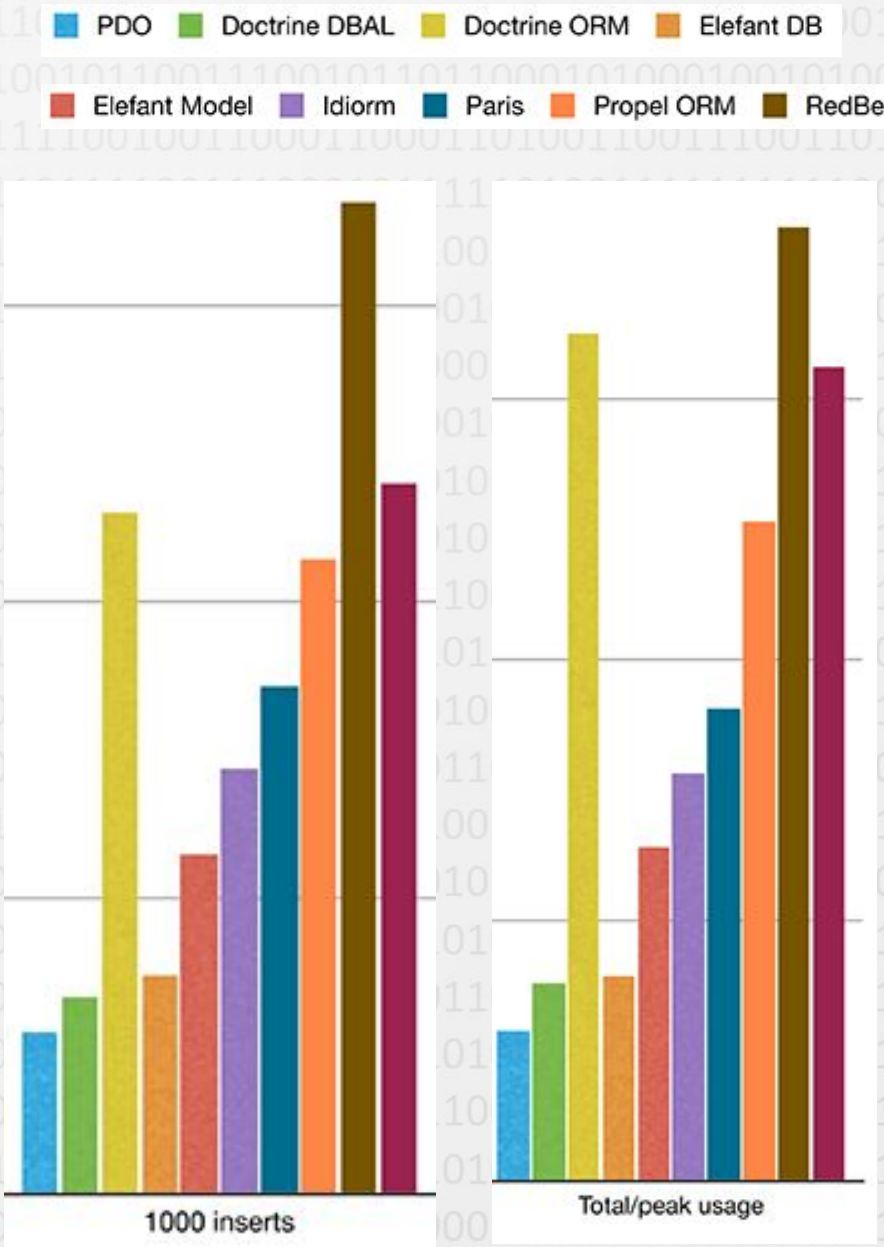
- **This is only one opinion, but there are typical pitfalls:**
  - Harder to configure
  - Bigger memory load
  - Bigger CPU load
  - Weak/no support for very complex queries/methods
  - Harder to optimize

It WILL have some overhead, we have to take it into consideration if it’s worth it or not (depending on the project we’re workin on).

ORM speed (C#)


SqlBulkCopy, 206.5
Simple.Data 1.0.0-rc0, 280.25
PetaPoco 4.03, 1388.5
Nhibernate 3.2.0.4, 1445.75
Insert
Linq2SQL, 4140.5
Entity Framework 5.0.0.0-Beta2, 2698.75
Entity Framework 4.1.10331.0, 2699.5
Dapper 1.8, 1040.25
0
500
1000
1500
2000
2500
3000
3500
4000
4500

ORM speed (PHP)



Advantages of using an ORM

- **In the code we don't use dialect-dependent SQL statements**
  - We can always switch dialect/server without changing the code
  - Using a self-made EF/Linq provider we can implement any storage methods
- **Instead of using string SQL statements, we use a format that is syntax-checked by the compiler before execution**
  - All syntax errors are revealed during development time
- **Instead of using string SQL parameters (string interpolation) we can use variables as query parameters**
  - Avoid SQL injection by design
- **The query results are not general object / object[] / associative arrays / typeless Dictionary collections**
  - Instead, we receive known strongly typed instances / collections **(!!!)**
- **By adding +1 layer on top of the ORM, we can have a data layer that can help us in exchanging the data layer and test the logic**
  - Repository with Dependency Injection

+1 layer ???

- **It is an advantage that the same LINQ query can be executed**
  - Independent from the actual data storage (List<T>, XML, MySQL/Oracle/MSSQL Database …)
  - Easier to separate the data processing code from the business logic code

- **Our aim: the LINQ query should refer to the data source so that it doesn't have to use the actual physical database**
  - Aim: write a Logic code where the actual data source can be defined during run-time
  - Aim: write a Logic code, where it's easy to write truly good unit tests (we want to test the logic without having a working data access layer)
  - *(Maybe solution, but not this semester: EF Core In-Memory Providers)*

- **This is what we need in the project work!!!**
  - Actual solution: Repository Design Pattern + Moq

Entity Framework versions

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

- **Linq To SQL: Similar syntax, „Not a complete implementation”**
- **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) the weird ObjectContext api is replaced with the great DbContext api!**
- **EF4.3 „Code First Migrations” (2012) a complete ORM!**
- **EF5 (2012), API changes …**
- **EF6 (2013), API changes …**
- **EF 6.1 (2014), EF 6.2 (2017), EF 6.4 (2019) widespread and GOOD!**

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, we reached the level of EF 6.1
- **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/
- **Currently:**
  EF 6.4 □ old codebase
  EF Core 3.1 □ In progress towards .NET Framework 5
- **https://docs.microsoft.com/en-us/ef/efcore-and-ef6/**
  - Almost all functionality is available, smaller missing features, good additions
  - 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

- **For every table we need a class with “similar structure”: Entity class / Entity model**
  - The essence of the „Data Mapper” is that tables and classes can be different
  - N:M relations will have an „entityless” mode: connector table without entity
  - In this semester, we shouldn’t try these:
    1 table = 1 entity class, SQL data fields = C# properties
- **DbContext (Model / Entity Model)**
  - The class that represents the database itself
  - It must handle the Connection and encapsulate the tables
  - Our own database will be handled by a descendent class
  - OnConfiguring() method to configure the connection itself
  - OnModelCreating() method to setup tables/entities:
    FLUENT API
- **DbSet<T> : IQueryable<T>**
  - Class to represent a table (= a collection of instances typed T)
  - In the DbContext there is a single property for all tables

Fluent API vs Annotations

- **Common aim: we want to use C# code to define that the C# properties in the entity classes map to which SQL data fields, including the occasional SQL-specific extras**
- **Annotations / Attributes: we extend the properties of the entity class with specific attributes**
  - Cars + Brands example

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

Fluent API vs Annotations

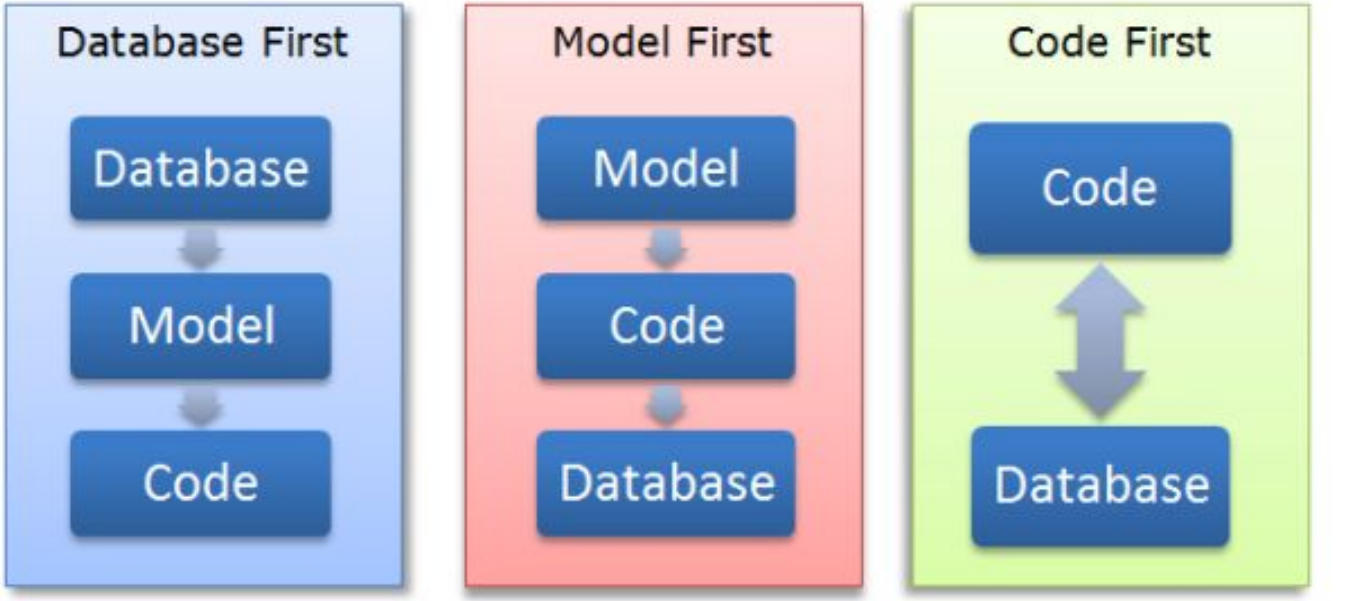
- **Fluent API: In the OnModelCreating() method of the DbContext descendent class**
  - More modern, better suited for the EF Core philosophy
  - EmpDept example
  - MUCH BETTER for foreign keys (With…Has…)
  - not all FluentAPI "functions" are available as attributes
  - but all attribute "functions" are available as FluentAPI

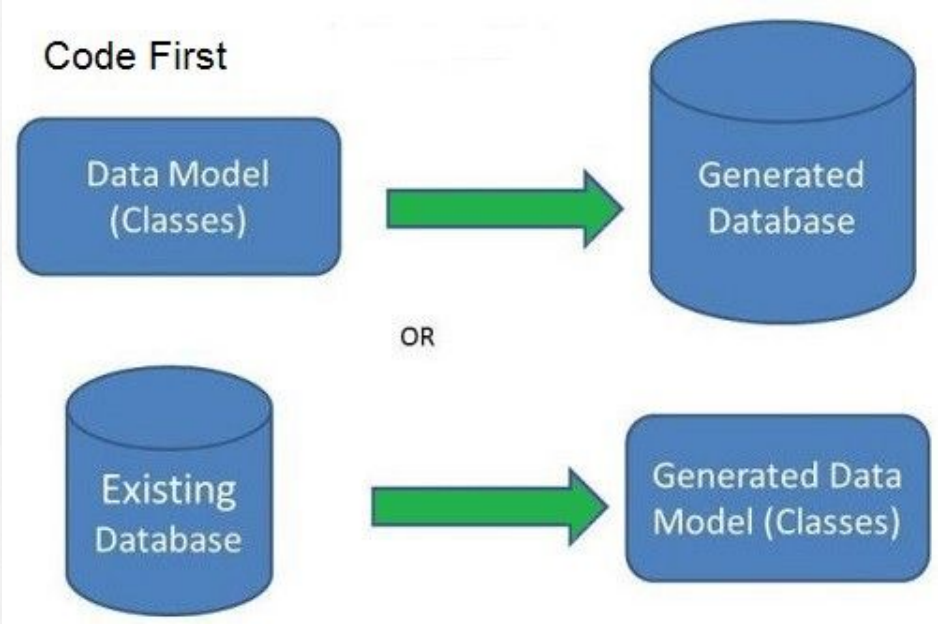
```
entity.HasKey(car => car.Id)
    .HasName("CAR_PRIMARY_KEY");

entity.Property(car => car.Id)
    .UseIdentityColumn() // IDENTITY
    .HasColumnName("car_id")
    .HasColumnType("int");

entity.ToTable("cars");
```

- **In the project work it doesn't matter which one we use**
  - Even if using attributes, we will need the OnModelCreating()
  - Because of Foreign Keys and Database Seeding


Database First
Database
Model
Code
Model First
Model
Code
Database
Code First
Code
Database


Code First
Data Model (Classes)
Generated Database
OR
Existing Database
Generated Data Model (Classes)

Generating the model (OLD SQL first)

SQL first
(we'll prefer code first!)

- **Pre-requirement: EF 6.x and pre-existing tables**
  http://msdn.microsoft.com/en-us/data/jj206878
- **Project, Add new Item, ADO.NET Entity Data Model <ADD>**
  - Generate from database <NEXT>
  - Have the MDF file in the drop-down + save connection settings <NEXT>;
  - EF6.0 <NEXT>;
  - Checkbox near all tables + Model namespace = EmpDeptModel <FINISH>
- **Depending on config: Template can harm your computer, click ok to run ... <OK> <OK>**
- **Result: automatically generated strongly typed classes, these are either typed versions of generic classes or rather POCO classes ~30KB for a two-table database (DataSet: 10x larger)**

Generating the model (NEW EF Core SQL first)

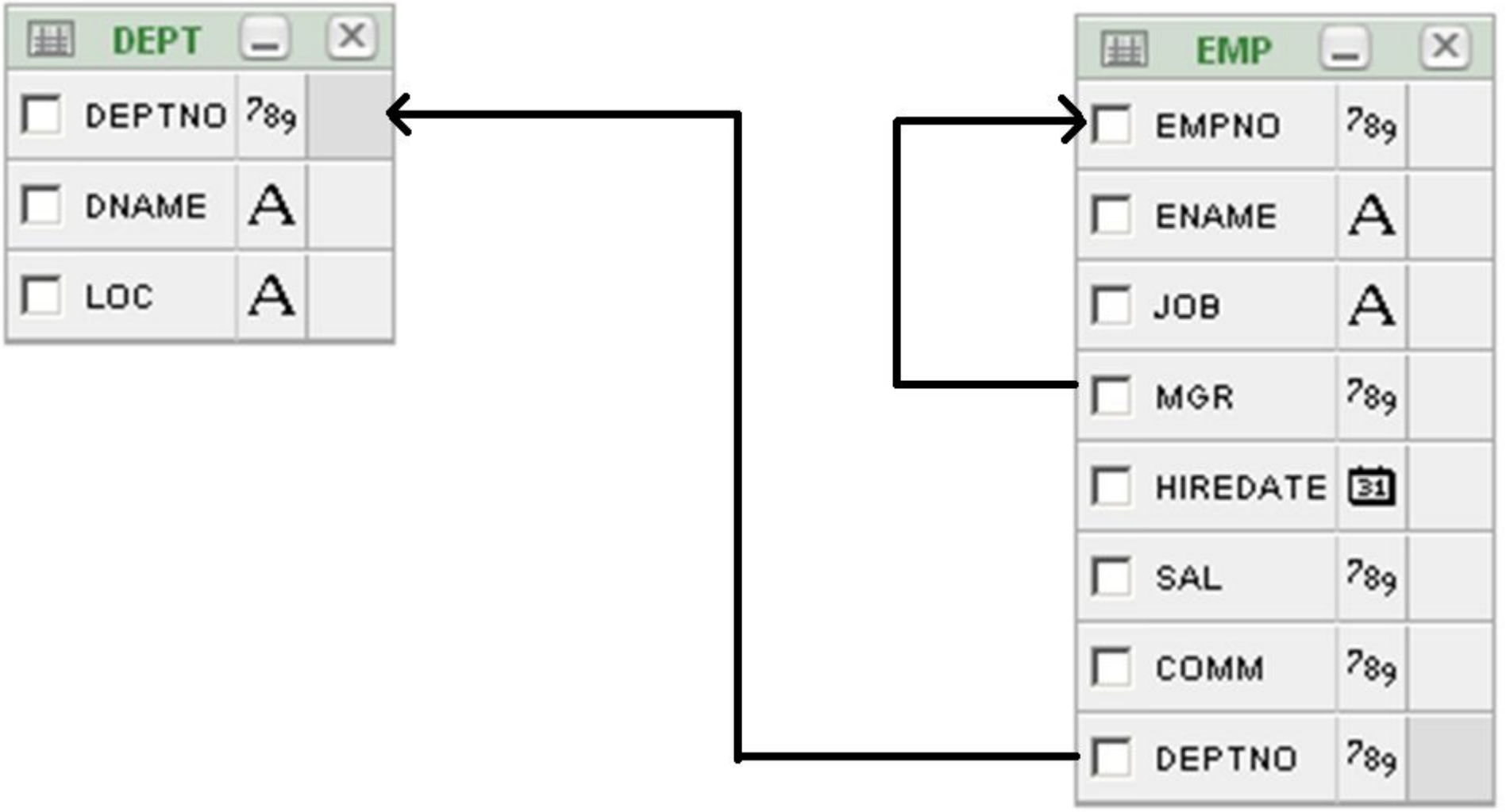
SQL first
(we'll prefer code first!)

- **EmpDept.mdf; pre-existing tables; Content+Copy Always**
- **Server Explorer EmpDept Right click Properties**
  - "Data Source = (LocalDB)\MSSQLLocalDB; AttachDbFilename = |DataDirectory|\EmpDept.mdf; Integrated Security = True"
  - „Close Connection" is important in the Server Explorer!!!
- **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
- **When using Code first approach, exactly the same classes should be written manually as the generated ones by the Scaffold-DbContext**
  - We need blank MDF+LDF files, and the connection string

Code First approach

1. **Create DataBase**
   - right click on project at solution explorer
   - add new item → Data → Service-based Database
   - .mdf and .ldf files created
   - .mdf and .ldf right click → properties → set *content* and *copy always*
   - rebuild solution → check in bin/debug/dotnetcore... folder for mdf and ldf files
   - Server Explorer → right click on database.mdf → copy connection string
   - right click again on database.mdf → close connection
   - change absolute path in the Connection String to relative path
2. **Used NuGet packages:**
   - EntityFramework.Core
   - EntityFramework.Tools
   - EntityFramework.Proxies
   - EntityFramework.SqlServer
3. **Create classes**
   - entity classes
   - DbConext class

Example Tables



1. Initialization

```
ED = new EmpDeptEntities();
Console.WriteLine("Connect OK");

// DbSet<T> type, with LINQ extension methods
// Can use LINQ query syntax too
// 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: Better performance, BUT we must know the connections that we'll use later and we *must reference* them when querying the data**

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

- **Lazy Loading: Weaker performance (especially if MANY records), but A LOT more comfortable to use, as we don't have to reference anything when querying the data**

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

  - We need UseLazyLoadingProxies() in the OnConfiguring() and then either we have to iterate over the .ToList() or we need to put „MultipleActiveResultSets = true" into the connection string

Example: in the EmpDept database…

1. Create and fill the database, then use SQL First approach to generate the model
   Examine the generated Entity and DbContext classes
2. List the workers names with their departments name
3. List the workers name, salary, job, and the job's average income (income = Sal + Comm)
4. List the workers name, their departments name, and the number of workers in that department
5. Double the salary of the company leader (PRESIDENT)
6. Delete those who joined the company less than 30 days after the hiredate of the president

Practice: Cars + Brands database creation

1. **Create the database, and then use fully Code First approach to create the tables and fill the data**
   (use annotations if you want – but the Fluent API is equally good)
2. **List all brands**
3. **List all cars, including the brand names (using Lazy Load)**
4. **List the cars, along with the average basePrice for every brand**