

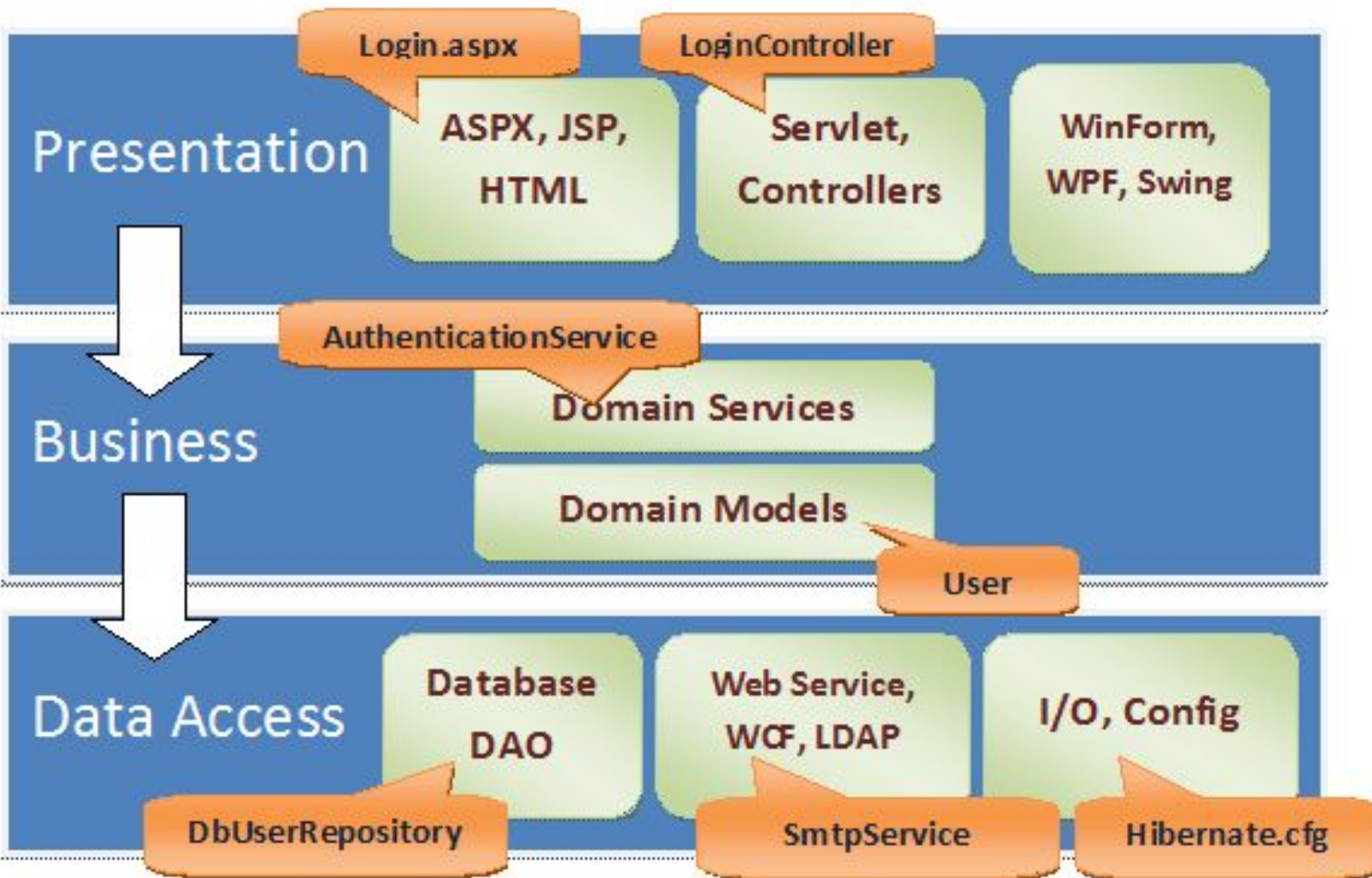
Advanced Development Techniques

Layering in our case

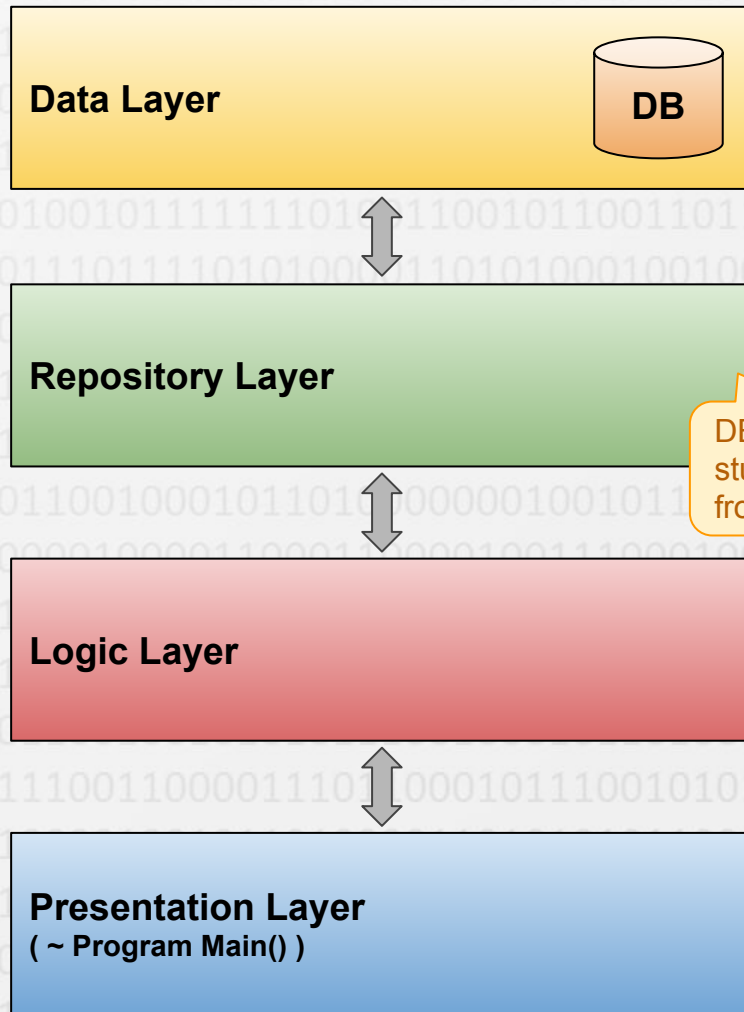
Project Work / Mid-Semester Exercise / FF

Typical Software Layers

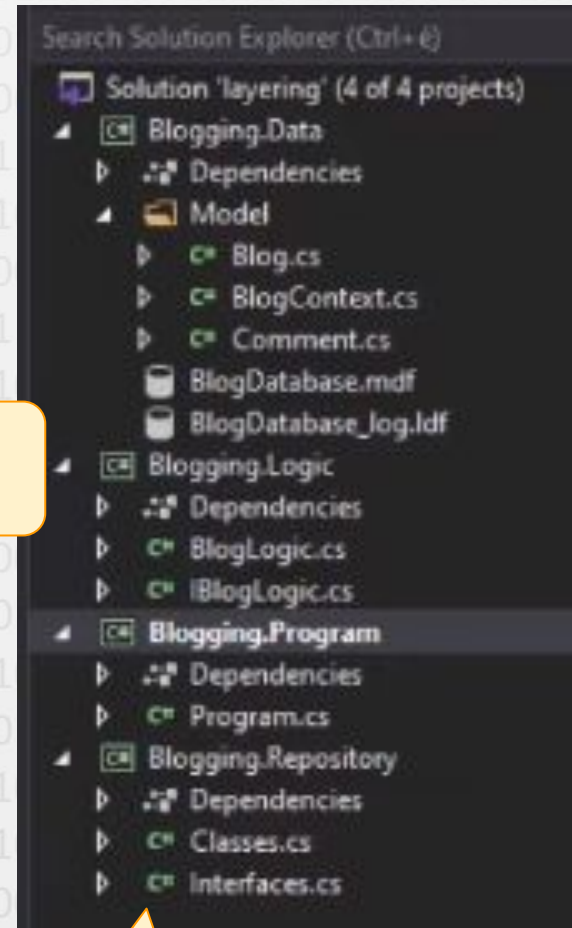
- All layers can be split up into layers/classes (SOLID!)
- Connecting points: using interfaces



Layering in our case

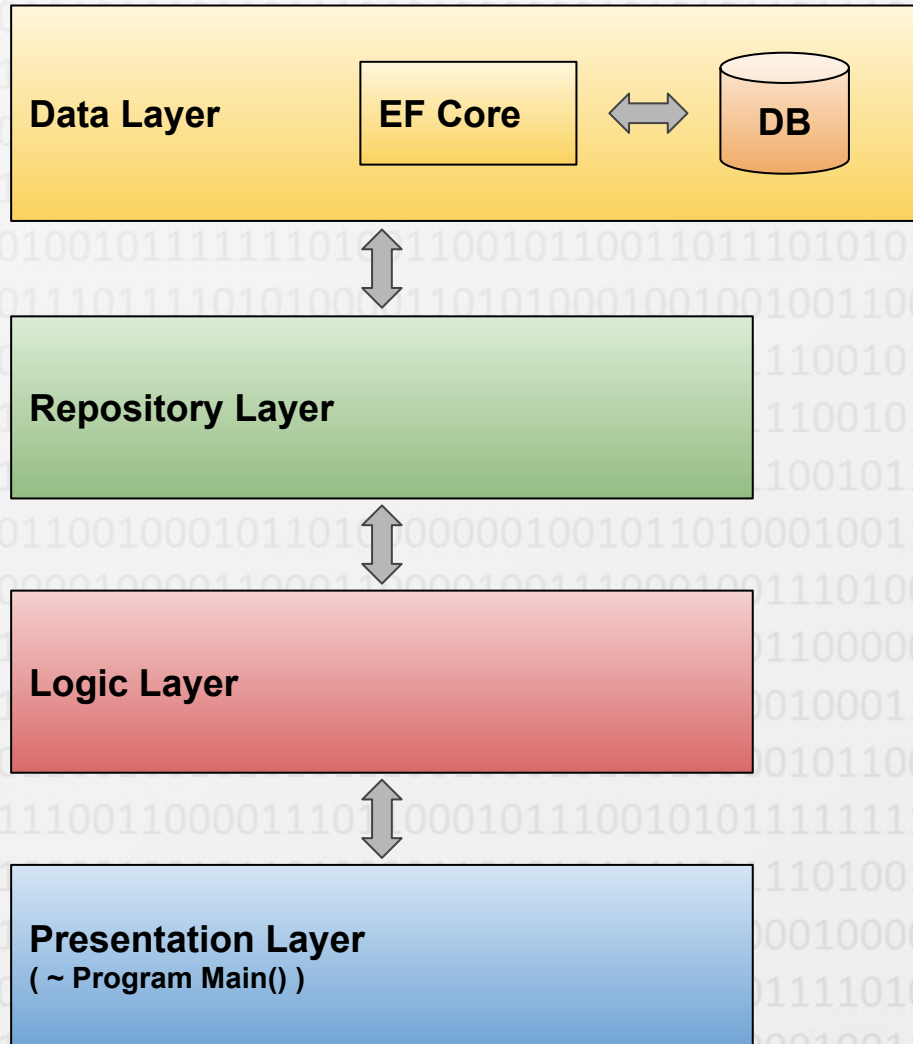


DB specific stuff is hidden from Logic!!!



Projects will be loaded as dll files.

Layering in our case



Layering structure

- **Layered project structure, at least 5 dotnet core C# projects**
 - **Data**: Database + Entity Framework to access it
 - **Repository**: EF data handler methods (IQueryable results)
 - **Logic**: BL methods that use one or more repository methods (CRUD + Non-Crud, the latter typically with LINQ queries, and list-typed results)
 - **Test**: Unit testable code (for the business logic methods, mocked Crud and mocked/asserted Non-Crud tests)
 - **Program**: Menu-driven (ConsoleMenu-simple, EasyConsole) console app, creates instances using a separated Factory class
 - this layer can be considered as the “UI”

Layers: DATA

```
public partial class CarDbContext : DbContext
{
    public virtual DbSet<Brand> Brand { get; set; }
    public virtual DbSet<Car> Cars { get; set; }

    public CarDbContext()
    {
        this.Database.EnsureCreated();
    }
}
```

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.
            UseLazyLoadingProxies().
            UseSqlServer(@"data source=(LocalDB)\MSSQLLocalDB;attachdbfilename=|
                DataDirectory|\CarDb.mdf;integrated
                security=True;MultipleActiveResultSets=True");
    }
}
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
```

Layers: REPOSITORY

```
public interface IRepository<T> where T : class

public interface ICarRepository : IRepository<Car>

public interface IBrandRepository : IRepository<Brand>
```

With the Repository layer the Logic layer will be testable! And also DI have to be used.

```
public abstract class Repository<T> : IRepository<T> where T : class
{
    protected DbContext ctx;
    public Repository(DbContext ctx)
    {
        this.ctx = ctx;
    }
    public IQueryable<T> GetAll()
    {
        return ctx.Set<T>(); // "Set" as a noun, not as a verb!!!
    }
}
```

The aim here is to place every general operation which is possible into a base class.

Modify usually not part of this, because modify-ing should be entity-specific (modify only the name // age // etc, not the full object).



```
public class CarRepository : Repository<Car>, ICarRepository
{
    public CarRepository(DbContext ctx) : base(ctx) { }
```

```
public class BrandRepository : Repository<Brand>, IBrandRepository
```

Layers: REPOSITORY

What can be written as completely generic, which will be true and can be applied to everything? → Not so much!

```
public abstract class EfRepository :  
    IRepository where TEntity : class  
{  
    DbContext context;  
    // context.Set<TEntity>() => the table that stores TEntity  
  
    public void AddNew(TEntity newInstance)  
        { /* generic implementation */ }  
    public void DeleteOld(TEntity oldInstance)  
        { /* generic implementation */ }  
    public IQueryable GetAll()  
        { /* generic implementation */ return null; }  
  
    public abstract TEntity GetById(int id);  
}
```



Layers: LOGIC – SOLID

```
// Should add DTO instead of Entities => SKIP for this semester
// This is a SERIOUS security hole!
// var car = logic.GetOneCar(40);
// car.Model = "NEW NAME"; // Shouldn't be able to do this.
// logic.ChangeCarPrice(40, car.car_baseprice); // saves new model name too!!!
```

```
public class AveragesResult
{
    public string BrandName { get; set; }
    public double AveragePrice { get; set; }
    public override string ToString()...
    public override bool Equals(object obj)...
    public override int GetHashCode()...
}
```

Entity classes can be reached / used from everywhere (every layer), but they shouldn't!!! This is absolutely not good, and should be avoided by using DTO classes, but it will be too much work, so we skip for this semester.

```
public interface ICarLogic
// Avoid: GOD OBJECT, too many responsibilities!
// SPLIT UP into multiple classes AS YOU WANT!
```

“Default” method overrides should be used because testing will need them.

```
{
    Car GetOneCar(int id);
    void ChangeCarPrice(int id, int newprice);
    IList<Car> GetAllCars();
    IList<AveragesResult> GetBrandAverages();
}
```

Avoid this!

Create classes (objects) to match the return types. Avoid object[] and string[] arrays/lists!

```
public class CarLogic : ICarLogic
```

Layers: LOGIC – SOLID + DI + CRUD

// Avoid god object => split up Logic into multiple classes

```
ICarRepository carRepo;
```

```
IBrandRepository brandRepo;
```

```
public CarLogic(ICarRepository carRepo, IBrandRepository brandRepo)
```

```
{  
    this.carRepo = carRepo;  
    this.brandRepo = brandRepo;  
}
```

Interface!
Important to use DI,
otherwise won't be
able to test Logic.

```
public void ChangeCarPrice(int id, int newprice)
```

```
{  
    carRepo.ChangePrice(id, newprice);  
}
```

```
public int AddBrand(string brandName)
```

```
{  
    return brandRepo.Add(brandName);  
}
```

```
public IList<Brand> GetAllBrands()
```

```
{  
    return brandRepo.GetAll().ToList();  
}
```

```
public IList<Car> GetCarsByBrand(int brand)
```

```
{  
    return carRepo.GetAll().Where(x => x.BrandId == brand).ToList();  
}
```


Layers: LOGIC – NON-CRUD

IList is a bit more flexible, so better to use interface references here as well.

```
public IList<AveragesResult> GetBrandAverages()
{
    var q = from car in carRepo.GetAll()
            group car by new { car.Brand.Id, car.Brand.Name } into grp
            select new AveragesResult()
            {
                BrandName = grp.Key.Name,
                AveragePrice = grp.Average(car => car.BasePrice) ?? 0
            };
    return q.ToList();
}

public IList<AveragesResult> GetBrandAveragesJoin()
{
    var q = from car in carRepo.GetAll()
            join brand in brandRepo.GetAll() on car.BrandId equals brand.Id
            let item = new { BrandName = brand.Name, Price = car.BasePrice }
            group item by item.BrandName into grp
            select new AveragesResult()
            {
                BrandName = grp.Key,
                AveragePrice = grp.Average(item => item.Price) ?? 0
            };
    return q.ToList();
}
```

Layers: TEST

```
[Test] // Noncrud test
public void TestGetAveragesJoin()
{
    var logic = CreateLogicWithMocks();
    var actualAverages = logic.GetBrandAveragesJoin();

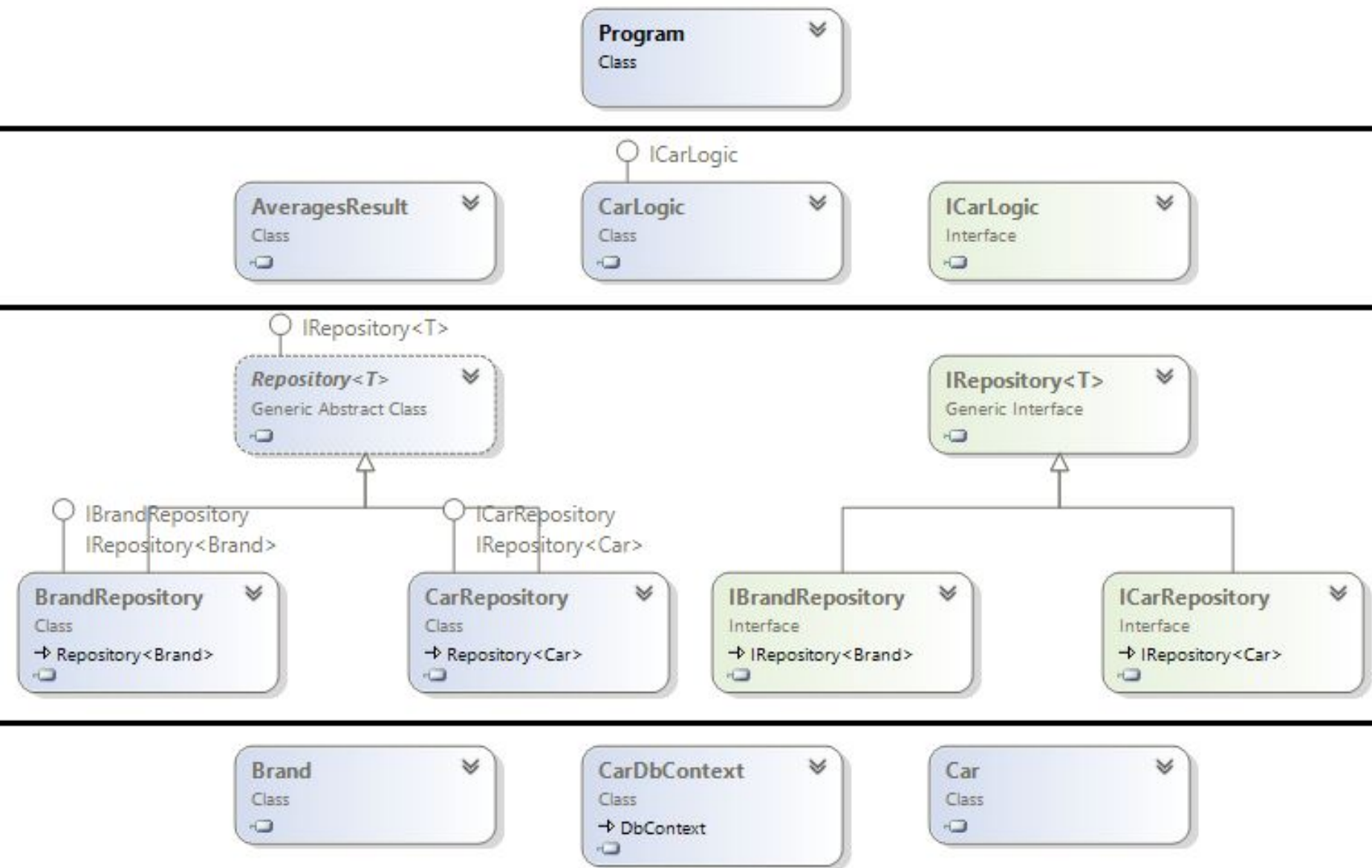
    Assert.That(actualAverages, Is.EquivalentTo(expectedAverages));
    carRepo.Verify(repo => repo.GetAll(), Times.Once);
    brandRepo.Verify(repo => repo.GetAll(), Times.Once);
}

[Test] // Crud Test: Add
public void TestBrandAdd()
{
    var logic = CreateLogicWithMocks();
    int idNumber = logic.AddBrand("Suzuki");
    Assert.That(idNumber, Is.EqualTo(42));
    brandRepo.Verify(repo => repo.Add(It.IsAny<string>()), Times.Once);
}
```


Layers: PROGRAM (~UI / Frontend)

- The Console App can **only** call Logic methods, the Logic will forward the CRUD operations to the Repo, and the Repo calls the DbContext methods
 - Every layer MUST only communicate with the layer directly below (occasional upwards communication: with events, not needed now)
- All DbContext/Repository/Logic instance creation is done here, preferably using a separated Factory class
 - Only ONE instance from the DbContext descendant will be used by all Repositories (Singleton Design Pattern is possible, for caveats see SGUI (prog4))!
 - The Logic classes use the same Repo instances from the Repo types
- Complies with the SOLID principles
 - **NO** other project than the Console App can use Console.Read/Write operations
 - The Logic and the Console App **MUST NOT USE** dbContext methods, only the Repository can do so
- Menu-driven: ConsoleMenu-simple / EasyConsole

Layered application



Problems between layers I.

- It can be problematic if the ORM Entity instances are available above the Data/Repo layers
 - E.g. we cannot modify a worker's salary in the Logic, but there is a method to relocate a department
 - ```
EMP singleEmp = myLogic.GetEmp(7788);
singleEmp.SAL = 10000; ← problematic line
myLogic.RelocateDept(30, "Budapest");
```
  - The SaveChanges() in the RelocateDept() will save the modified salary!
  - This is not a problem in this project work, but in real-world applications you should use intermediate Business Model / DTO classes
    - Business Model = if it contains some business related methods / functions
    - DTO = data transfer object = if it only contains properties and it's values

# Problems between layers II.

- **Solutions for passing instances between layers:**
  - Pass functionally-restricted entity instances (e.g. without setters)
  - Visibility: inside a layer you can see the internals too, outside only the public
  - Manual conversion (reflection ... Performance?)
  - Automatic Conversion (AutoMapper ... Production-Ready?)



# Problems between layers III.

- **Signaling errors (layer-specific exception handling)**
  - Don't have to create exceptions for every possible solution
    - eg. email contains @ **AND** email contains dot **AND** email must be at least x length...
    - but try to check a few basic things like GetByID → exception if there is no such id, etc.
  - EF hides the layers below
    - SQL exceptions won't be displayed to you directly
    - You will see only EF exception (which is wrapped the SQL exception for example)
  - InnerException / AggregateException

# Inner structure of layers

- **Split up layers**

- Repository ☐ stated exactly in the prev. slides
  - IBaseRepo, IEntity1Repo IEntity2Repo etc.
- Logic ☐ use your brain, split it up somehow\* and “it will be ok”
  - Ok, but how?!?!
    - Split up based on entities → too fragmented
    - Split up based on functions → better way
  - Hard to explain...

- **Minimal requirements**

- Avoid Spagetti code / Big Ball Of Mud ☐ that's why we use layering
- EF + IRepository<T>/IEmpRepository + ILogic/IEmpLogic + Console

- **Problem: the Repository can be good, but the Logic...**

- God Object: Too many responsibilities for a single class
  - Typical symptom: too long class
  - Typical symptom: class with many dependencies / ctor parameters
  - Solution: Refactor to individual classes

# Inner structure of layers

- **EmpLogic, DeptLogic, PremiumCalculationLogic ...**
  - Ravioli code: small and easy classes in theory, but they make understanding the system as a whole difficult (... Repository-pattern is AntiPattern?!)
    - It's good until we have not so many tables
    - table → entity
    - Having 30+ tables makes too much classes / interfaces / etc. → hard to deal with it
  - Lasagna Code: layered code at the first glance, but impossible to handle due to the inner chaos and crazy internal cross-references
  - **MAIN GOAL:** Find the middle ground ☐ **REUSEABLE CODE!**

