From ddd to DDD

data-driven development

Domain-driven Design

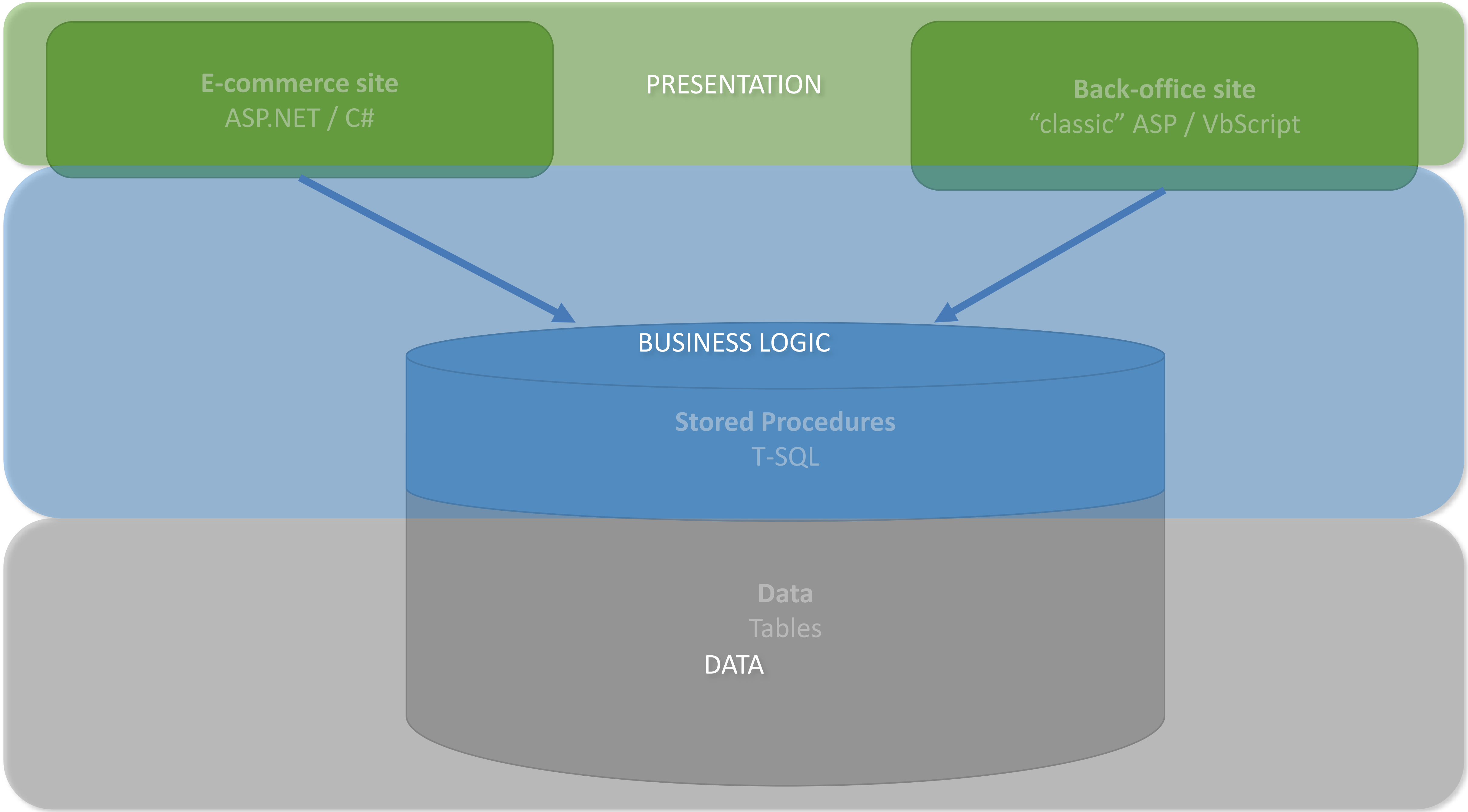
An ongoing journey

Thibaud Desodt - @tsimbalar

What this is about

- Me, myself and I
- Code
- Architecture
- Learning process

Where it all started

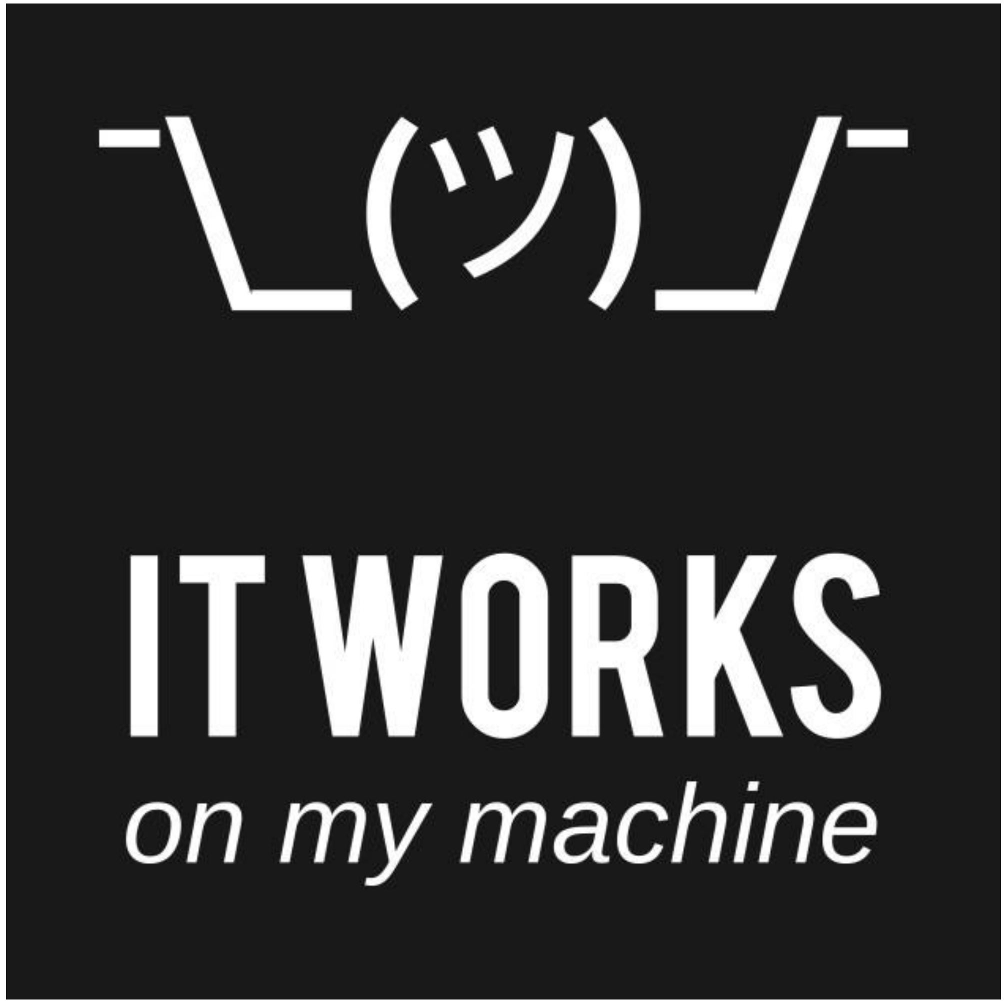

E-commerce site
ASP.NET / C#
PRESENTATION
Back-office site
"classic" ASP / VbScript
BUSINESS LOGIC
Stored Procedures
T-SQL
Data
Tables
DATA

And it was …

- Not super pleasant
  - VbScript
  - T-SQL
- Fragile
  - No automated tests
  - Tight coupling
    - Schema -> Stored Procs -> Website / admin site

… but it worked “well enough”!

- Customer value
  - Happy customer
  - Fast delivery of features
  - Reasonable perf
- Easy to work on
  - 1 feature ≈ 1 stored procedure



It worked well … in **that** context

- Working alone on project
- Version 1 of the product
- Well-defined requirements
- Tight interactions with customer
- Simple domain

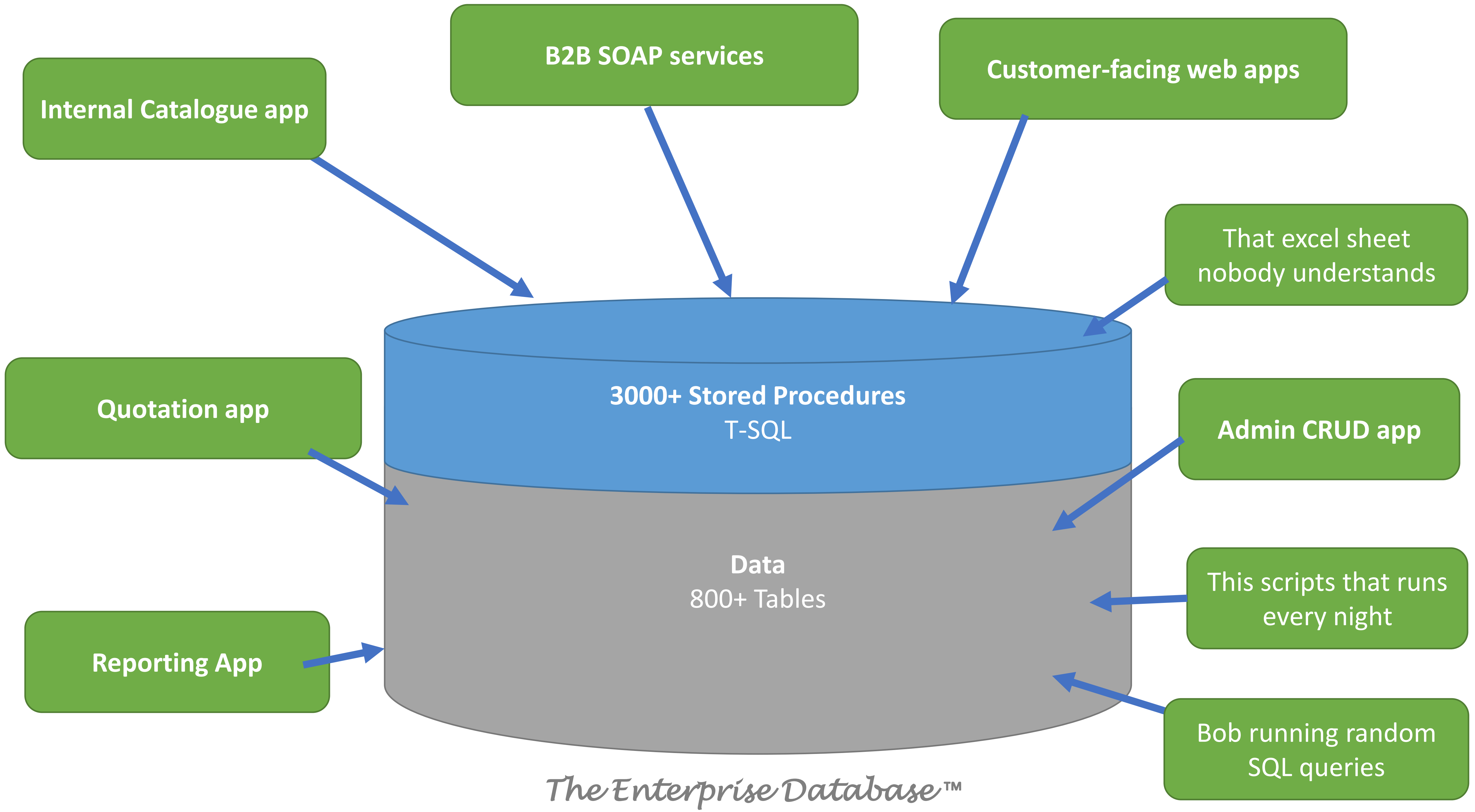
= ideal greenfield project

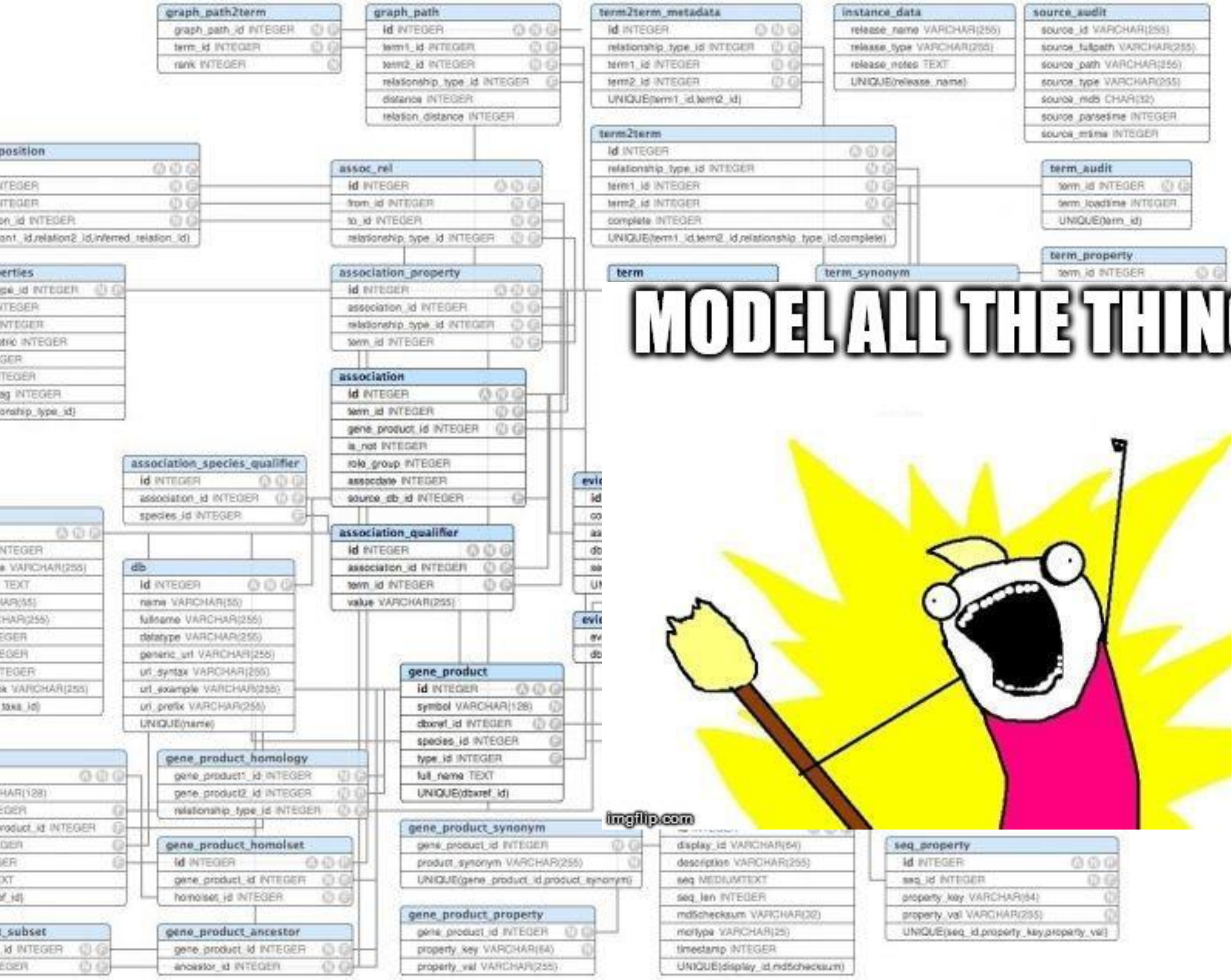
A few years later …

Similar approach, different context

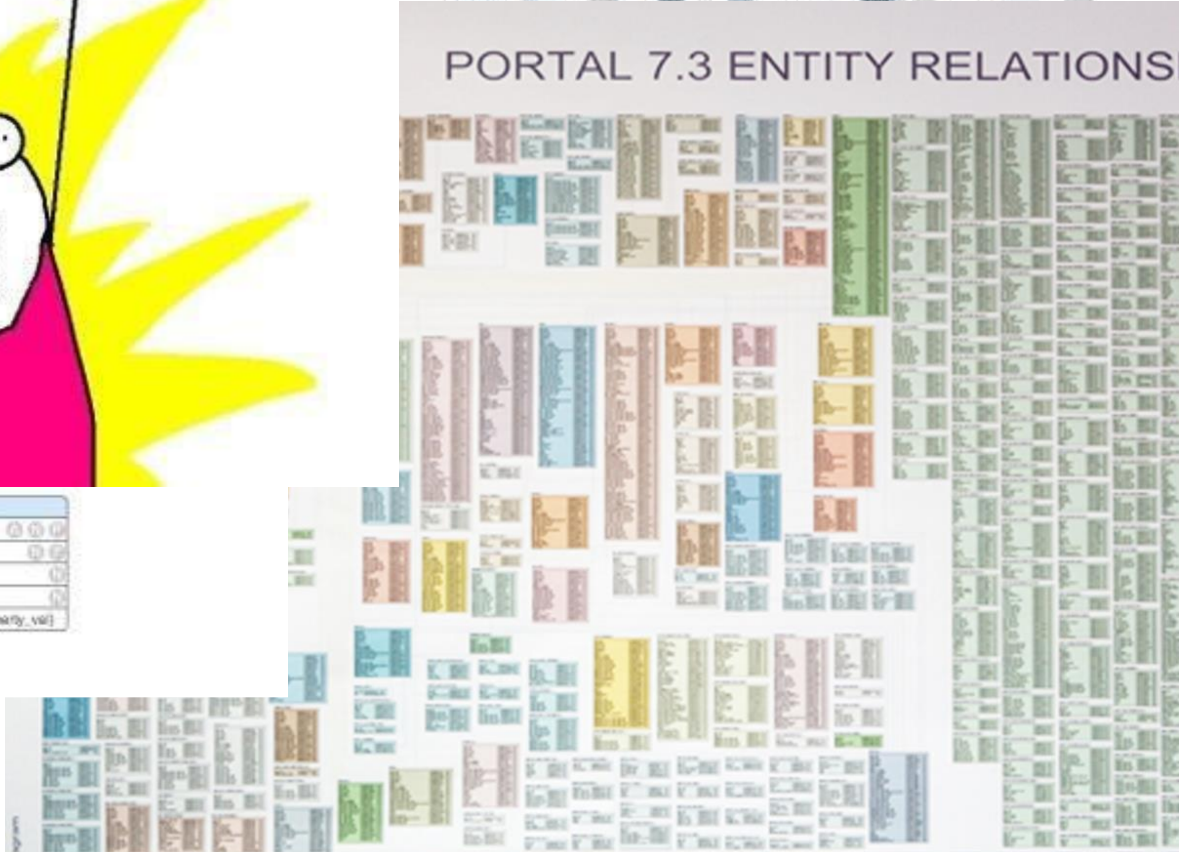
A different context

- 40,000+ employees company
- 100s of developers spread across the globe
- Many different systems accessing the database(s)


B2B SOAP services
Customer-facing web apps
Internal Catalogue app
That excel sheet nobody understands
Quotation app
3000+ Stored Procedures
T-SQL
Admin CRUD app
Data
800+ Tables
This scripts that runs every night
Reporting App
Bob running random SQL queries
The Enterprise Database™

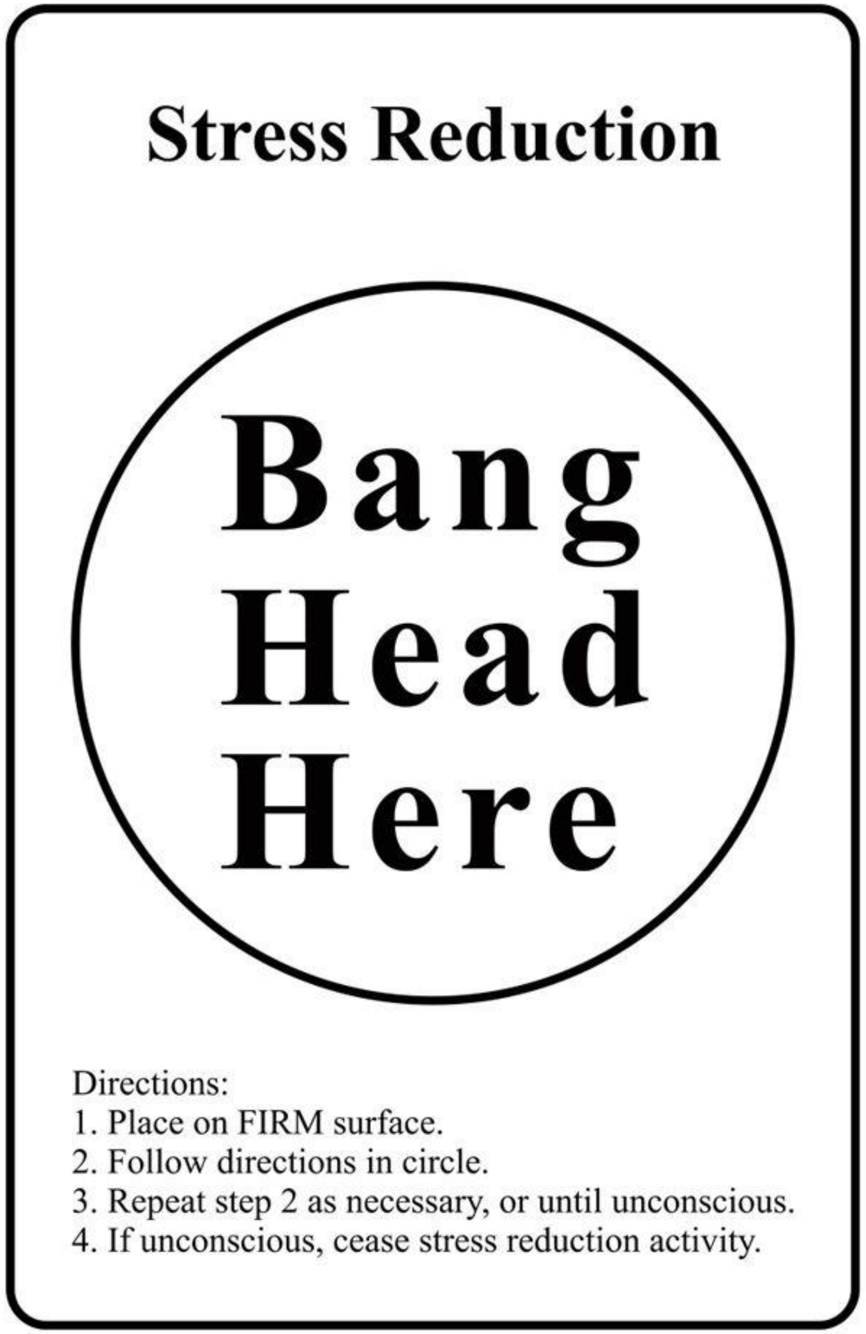

graph_path2term
graph_path
term2term_metadata
instance_data
source_audit
assoc_rel
term2term
term_audit
association_property
term_property
term
term_synonym
association
association_species_qualifier
association_qualifier
db
gene_product
gene_product_homology
gene_product_synonym
gene_product_homolset
seq_property
gene_product_property
gene_product_ancestor

MODEL ALL THE THINGS !
imgflip.com


PORTAL 7.3 ENTITY RELATIONS

Many issues

- Evolution is hard
- Testing is hard
- Versioning / collaboration is hard
- Performance is not great



Stress Reduction

Bang Head Here

Directions:
1. Place on FIRM surface.
2. Follow directions in circle.
3. Repeat step 2 as necessary, or until unconscious.
4. If unconscious, cease stress reduction activity.

the “solution” ….

- Team of 50 DBAs
- The “database” committee
- The “database” change process
- The “meta-database”
- Db replication
- Governance

Technical solutions
*… to solve **technical** issues*
*… introduced because of **technical** decisions*
*… with **no value** to the users*

*= **Accidental complexity***

How to Change Your Life
When You Feel STUCK
bigstockphoto.com/contributor=eelnosiva

Moving away from database-driven

- Persistence is an **implementation detail**

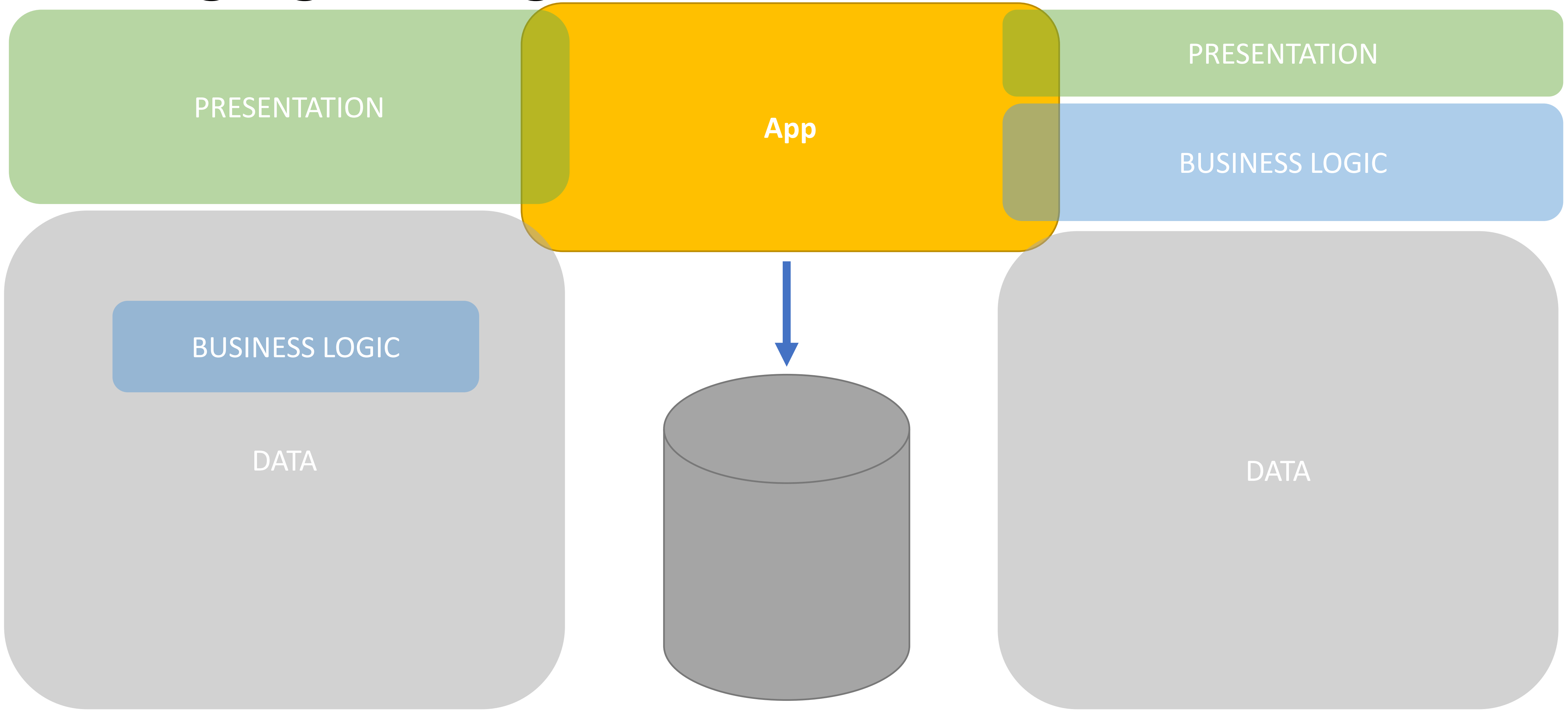
  Relational DB, Document DB, Key-Value store, file …

  Who cares ?

- Focus on **customer value**

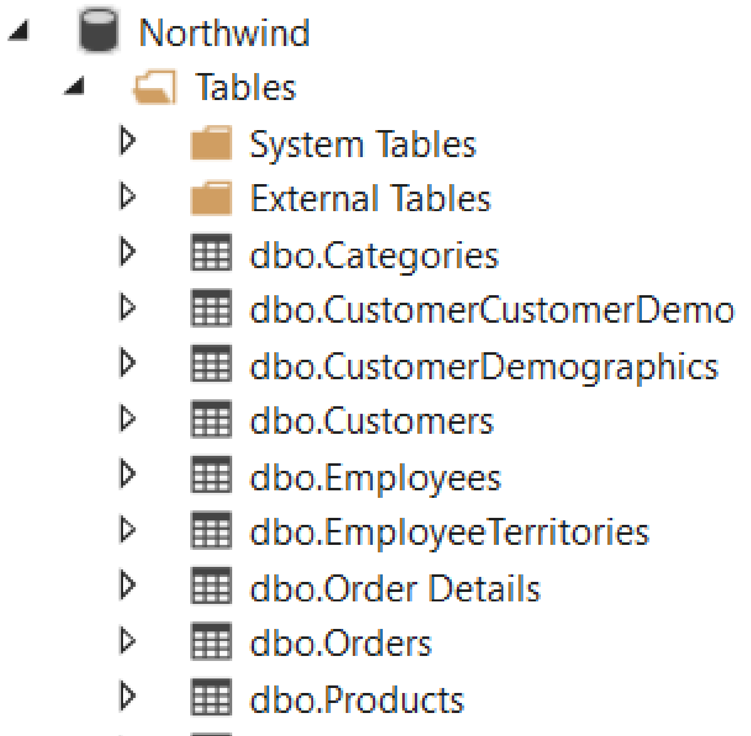
  Business > Tech

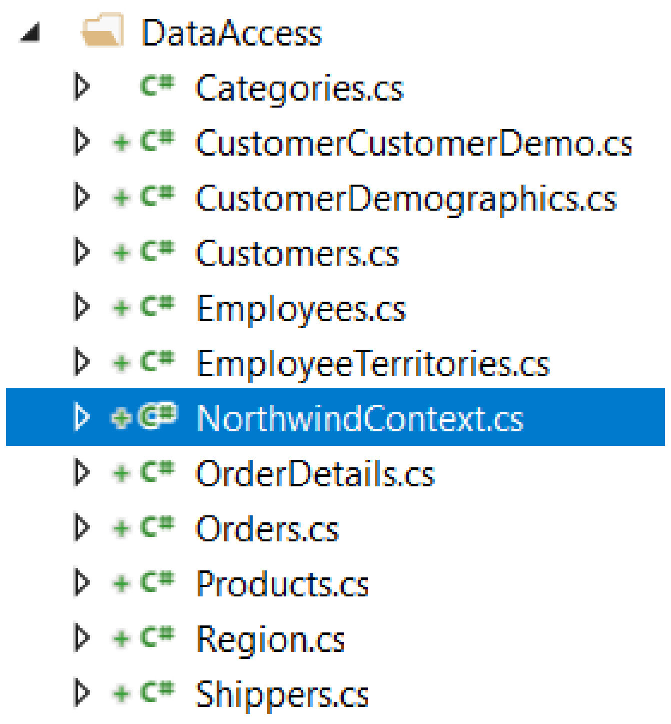
Bringing the logic back to the code



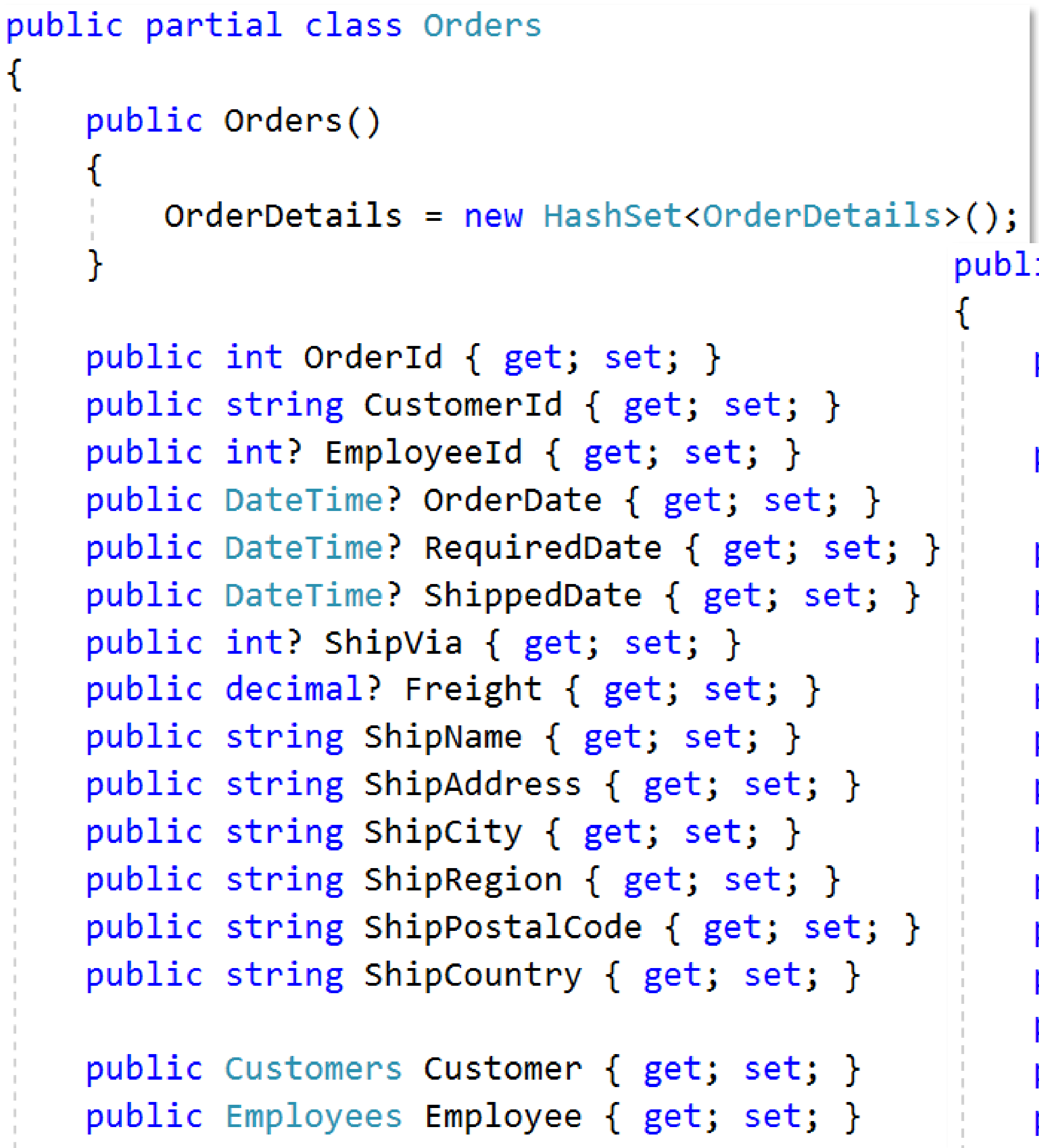
Bringing the logic back to the code

- **No more** Stored Procedures
- ORM (Entity Framework in that case)
- Data-access code + Entities* generated from database

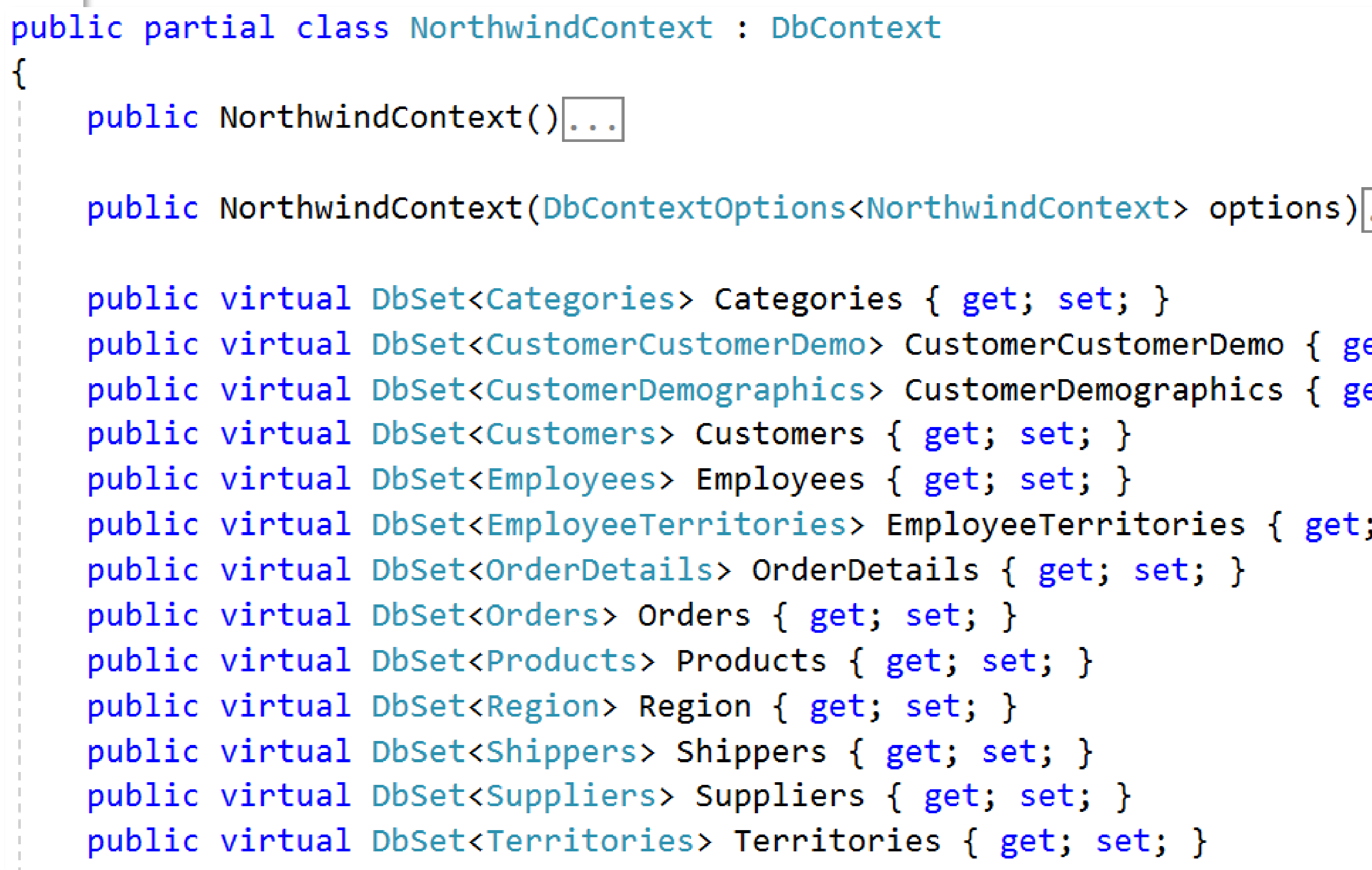

Northwind
Tables
System Tables
External Tables
dbo.Categories
dbo.CustomerCustomerDemo
dbo.CustomerDemographics
dbo.Customers
dbo.Employees
dbo.EmployeeTerritories
dbo.Order Details
dbo.Orders
dbo.Products


DataAccess
Categories.cs
CustomerCustomerDemo.cs
CustomerDemographics.cs
Customers.cs
Employees.cs
EmployeeTerritories.cs
NorthwindContext.cs
OrderDetails.cs
Orders.cs
Products.cs
Region.cs
Shippers.cs

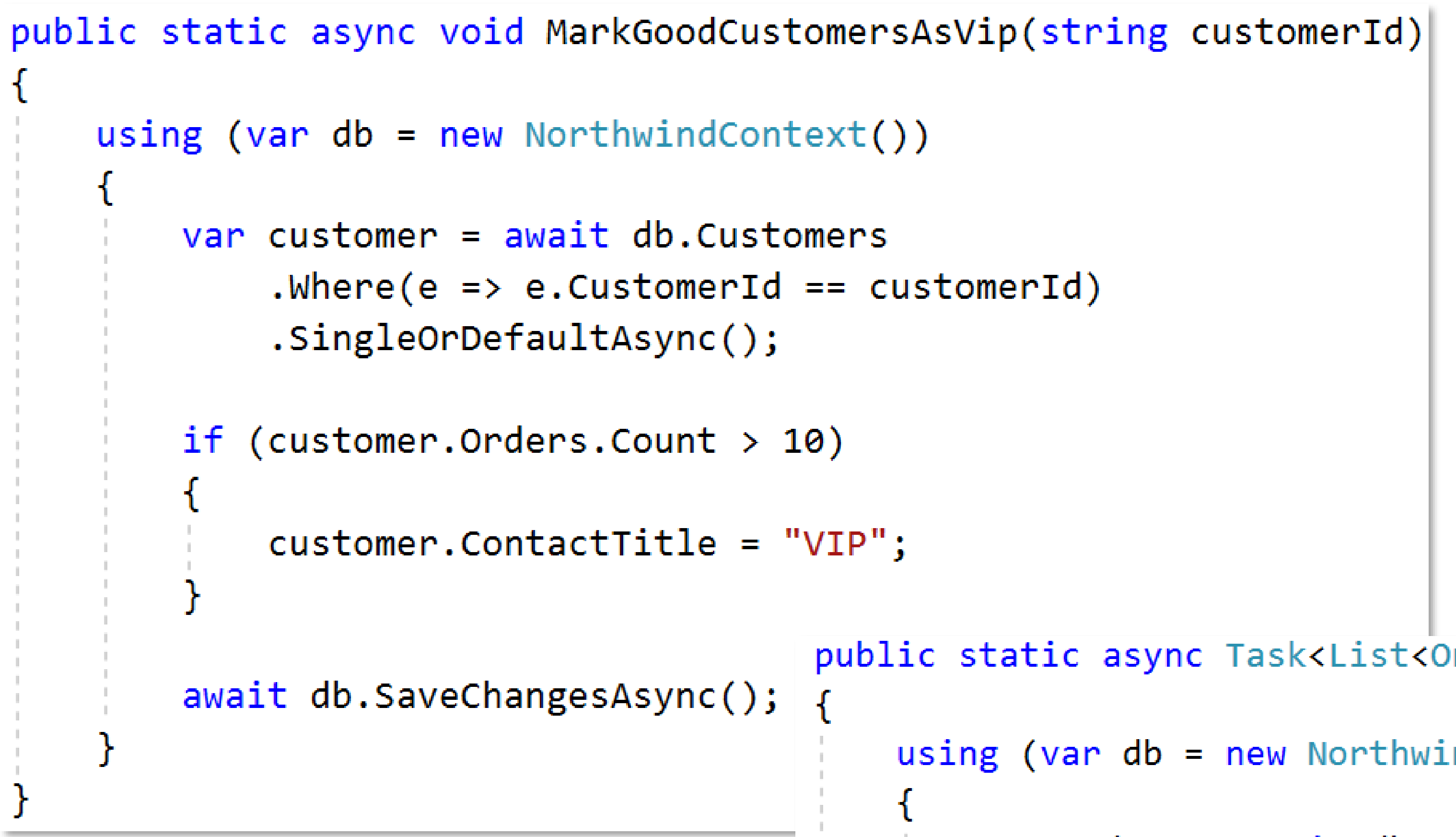
* Not quite



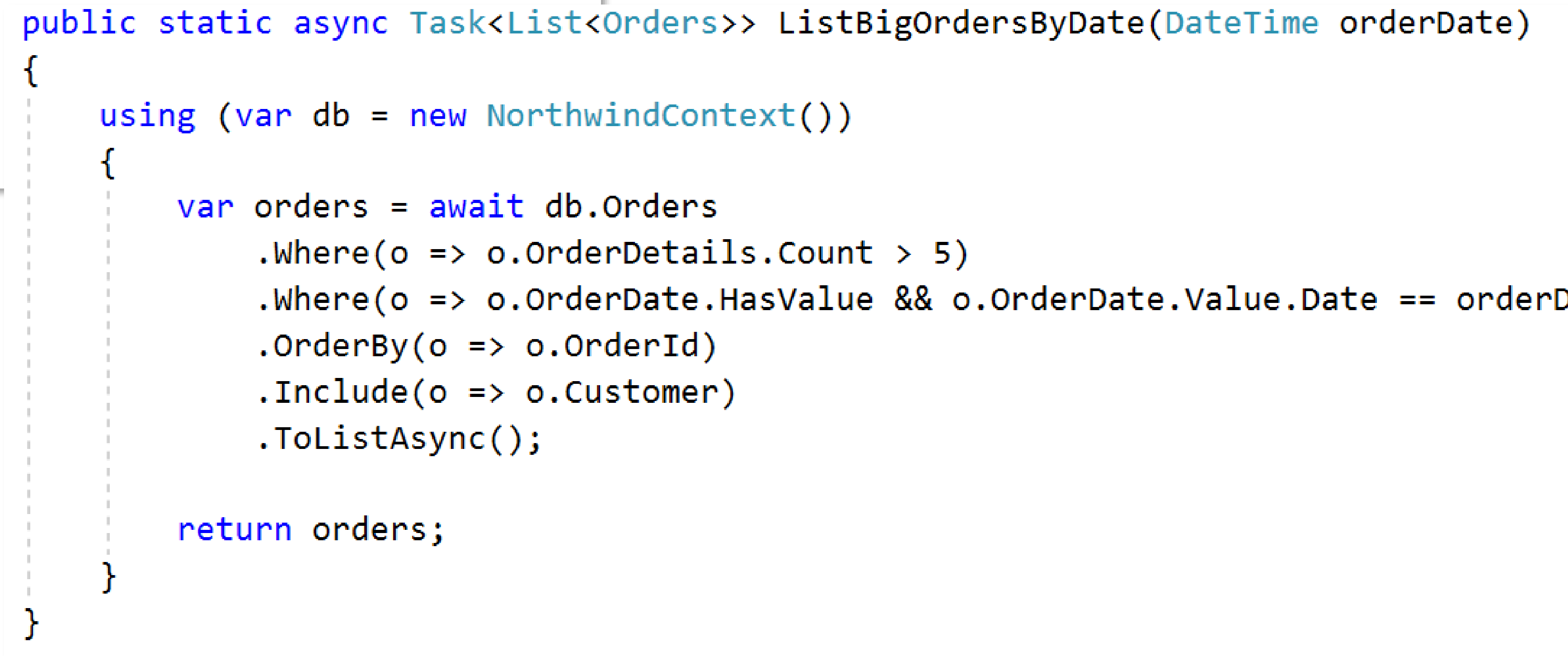
```csharp
public partial class Orders
{
    public Orders()
    {
        OrderDetails = new HashSet<OrderDetails>();
    }

    public int OrderId { get; set; }
    public string CustomerId { get; set; }
    public int? EmployeeId { get; set; }
    public DateTime? OrderDate { get; set; }
    public DateTime? RequiredDate { get; set; }
    public DateTime? ShippedDate { get; set; }
    public int? ShipVia { get; set; }
    public decimal? Freight { get; set; }
    public string ShipName { get; set; }
    public string ShipAddress { get; set; }
    public string ShipCity { get; set; }
    public string ShipRegion { get; set; }
    public string ShipPostalCode { get; set; }
    public string ShipCountry { get; set; }

    public Customers Customer { get; set; }
    public Employees Employee { get; set; }
```



```csharp
public partial class NorthwindContext : DbContext
{
    public NorthwindContext()...

    public NorthwindContext(DbContextOptions<NorthwindContext> options)

    public virtual DbSet<Categories> Categories { get; set; }
    public virtual DbSet<CustomerCustomerDemo> CustomerCustomerDemo { ge
    public virtual DbSet<CustomerDemographics> CustomerDemographics { ge
    public virtual DbSet<Customers> Customers { get; set; }
    public virtual DbSet<Employees> Employees { get; set; }
    public virtual DbSet<EmployeeTerritories> EmployeeTerritories { get;
    public virtual DbSet<OrderDetails> OrderDetails { get; set; }
    public virtual DbSet<Orders> Orders { get; set; }
    public virtual DbSet<Products> Products { get; set; }
    public virtual DbSet<Region> Region { get; set; }
    public virtual DbSet<Shippers> Shippers { get; set; }
    public virtual DbSet<Suppliers> Suppliers { get; set; }
    public virtual DbSet<Territories> Territories { get; set; }
```



```
public static async void MarkGoodCustomersAsVip(string customerId)
{
    using (var db = new NorthwindContext())
    {
        var customer = await db.Customers
            .Where(e => e.CustomerId == customerId)
            .SingleOrDefaultAsync();

        if (customer.Orders.Count > 10)
        {
            customer.ContactTitle = "VIP";
        }

        await db.SaveChangesAsync();
    }
}
```



```
public static async Task<List<Orders>> ListBigOrdersByDate(DateTime orderDate)
{
    using (var db = new NorthwindContext())
    {
        var orders = await db.Orders
            .Where(o => o.OrderDetails.Count > 5)
            .Where(o => o.OrderDate.HasValue && o.OrderDate.Value.Date == orderD
            .OrderBy(o => o.OrderId)
            .Include(o => o.Customer)
            .ToListAsync();

        return orders;
    }
}
```

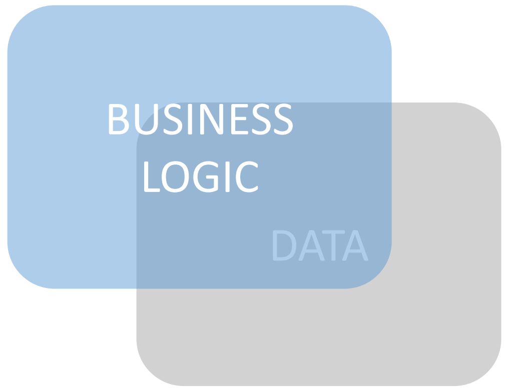
Status

That’s better !

- No more business logic in the DB
- Quite readable

Still a bit messy:

- Not testable
- Hard coupling



Better layering

Better layering

Services

BUSINESS LOGIC

Orchestrate data-flow for a given use-case

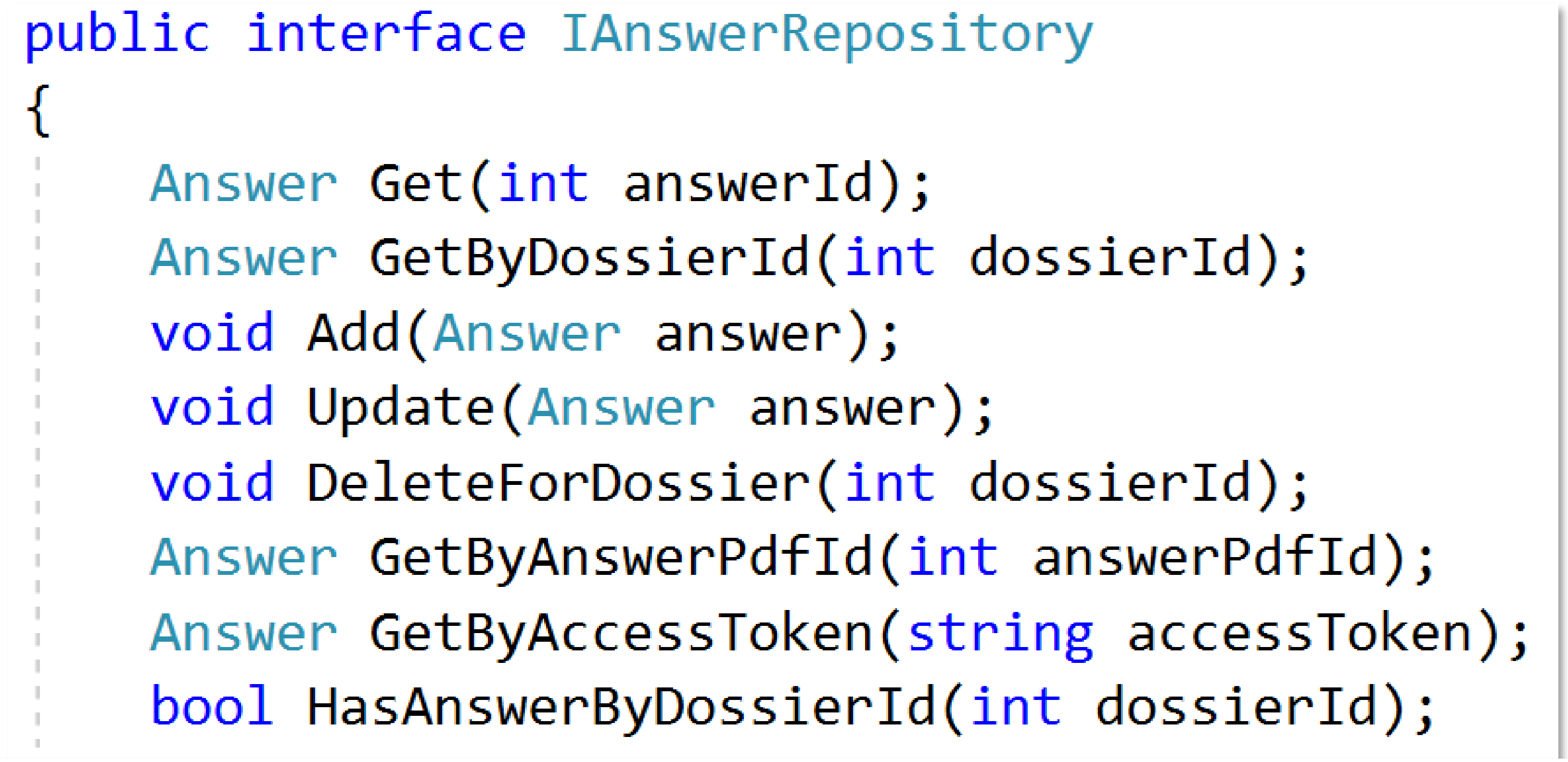
Repositories

+ Unit of Work

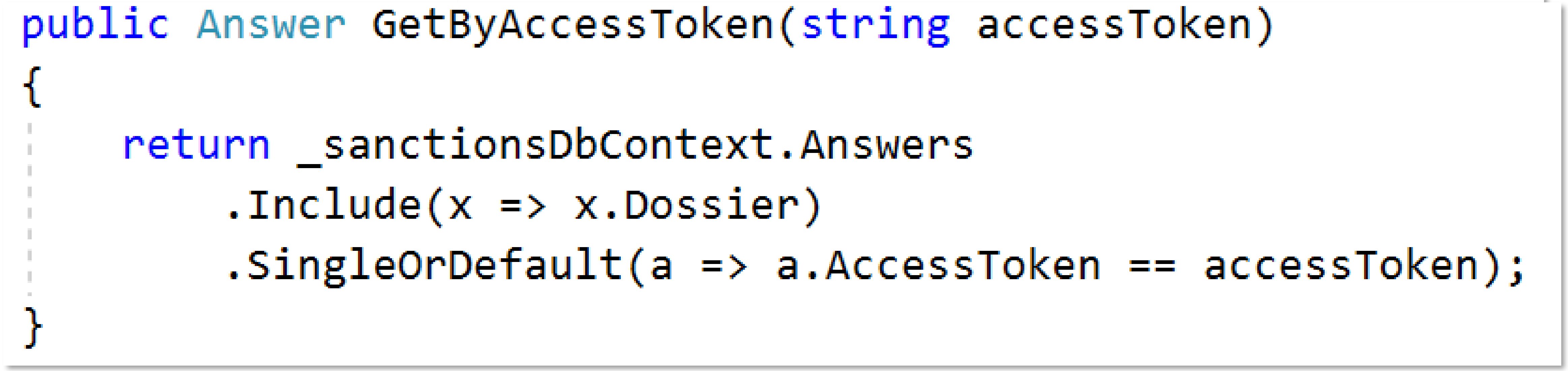
DATA

Abstract away details of how data is accessed

Repository

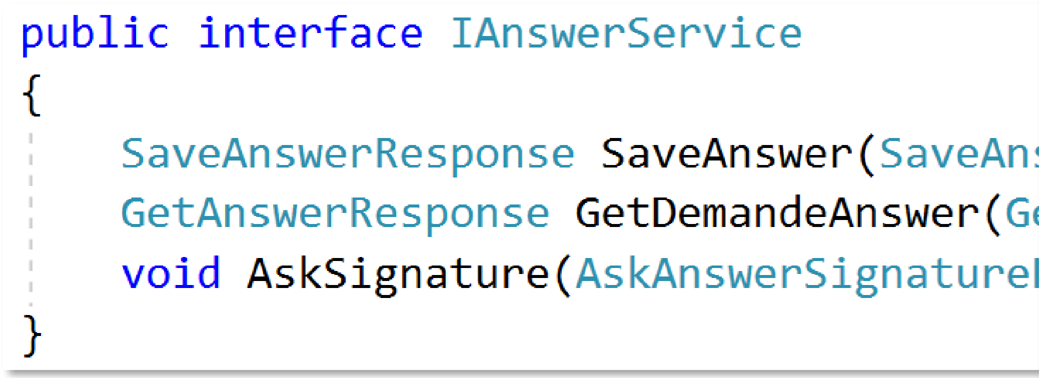


```
public interface IAnswerRepository
{
    Answer Get(int answerId);
    Answer GetByDossierId(int dossierId);
    void Add(Answer answer);
    void Update(Answer answer);
    void DeleteForDossier(int dossierId);
    Answer GetByAnswerPdfId(int answerPdfId);
    Answer GetByAccessToken(string accessToken);
    bool HasAnswerByDossierId(int dossierId);
```

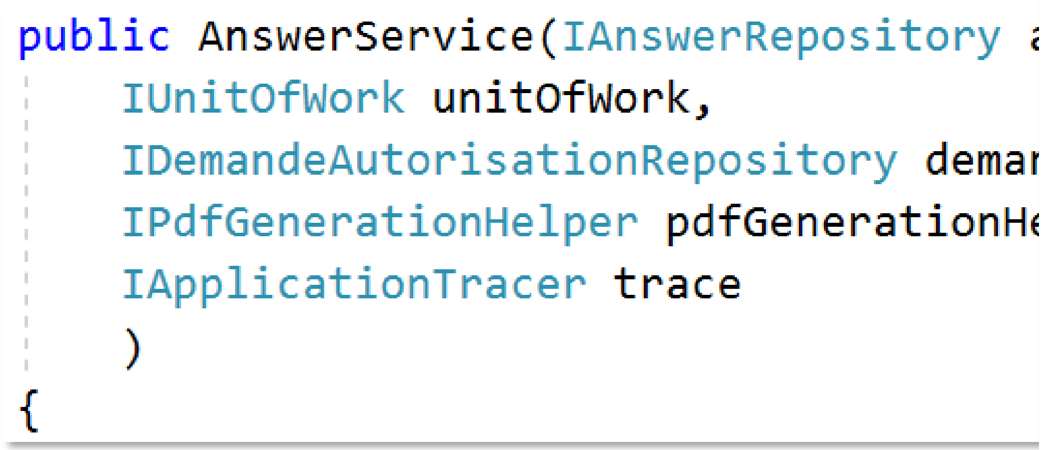


```
public Answer GetByAccessToken(string accessToken)
{
    return _sanctionsDbContext.Answers
        .Include(x => x.Dossier)
        .SingleOrDefault(a => a.AccessToken == accessToken);
}
```

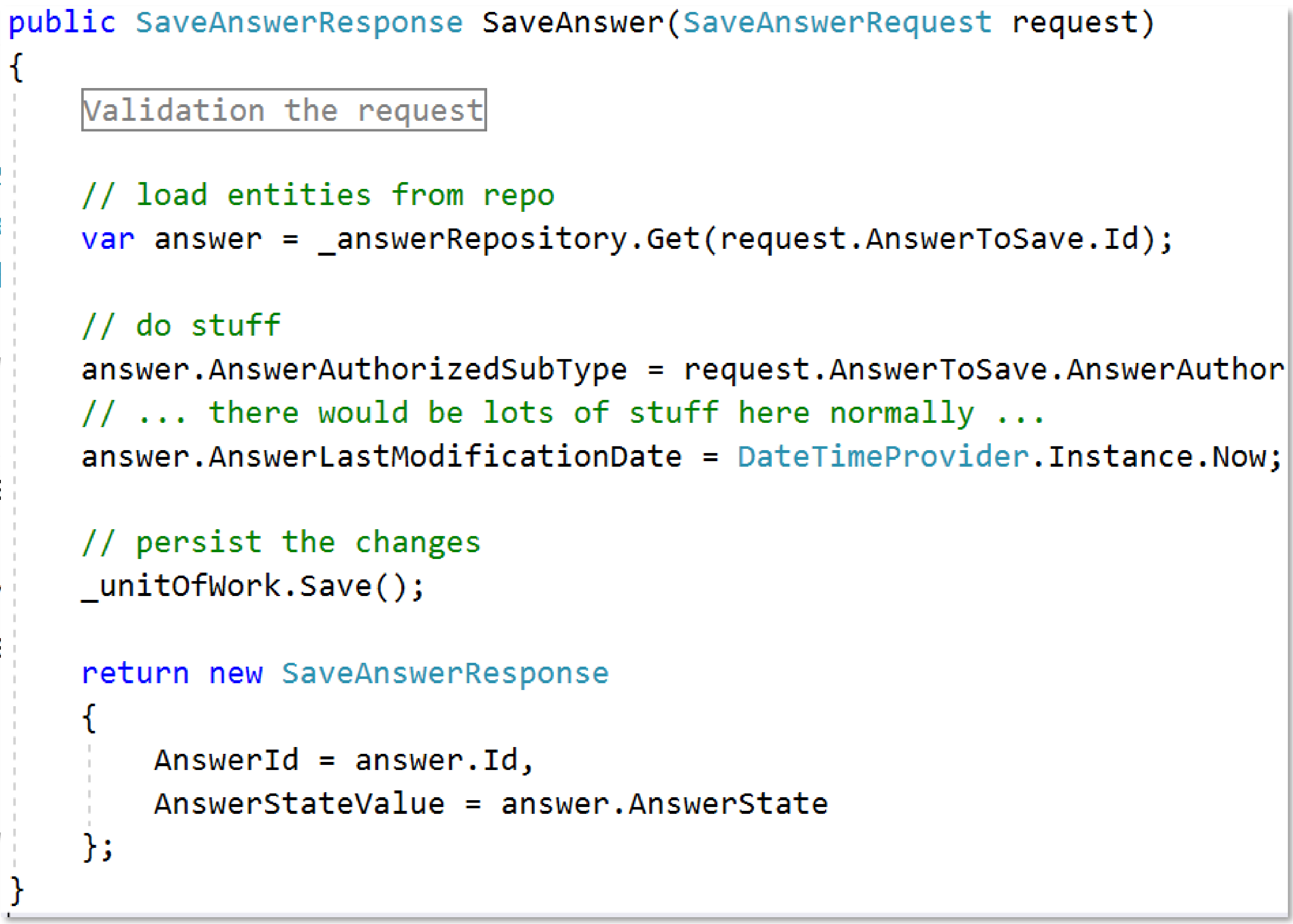
A “Service”



```
public interface IAnswerService
{
    SaveAnswerResponse SaveAnswer(SaveAns
    GetAnswerResponse GetDemandeAnswer(Ge
    void AskSignature(AskAnswerSignatureI
}
```



```
public AnswerService(IAnswerRepository a
    IUnitOfWork unitOfWork,
    IDemandeAutorisationRepository demar
    IPdfGenerationHelper pdfGenerationHe
    IApplicationTracer trace
    )
{
```



```
public SaveAnswerResponse SaveAnswer(SaveAnswerRequest request)
{
    Validation the request

    // load entities from repo
    var answer = _answerRepository.Get(request.AnswerToSave.Id);

    // do stuff
    answer.AnswerAuthorizedSubType = request.AnswerToSave.AnswerAuthor
    // ... there would be lots of stuff here normally ...
    answer.AnswerLastModificationDate = DateTimeProvider.Instance.Now;

    // persist the changes
    _unitOfWork.Save();

    return new SaveAnswerResponse
    {
        AnswerId = answer.Id,
        AnswerStateValue = answer.AnswerState
    };
}
```

Status

Quite an improvement

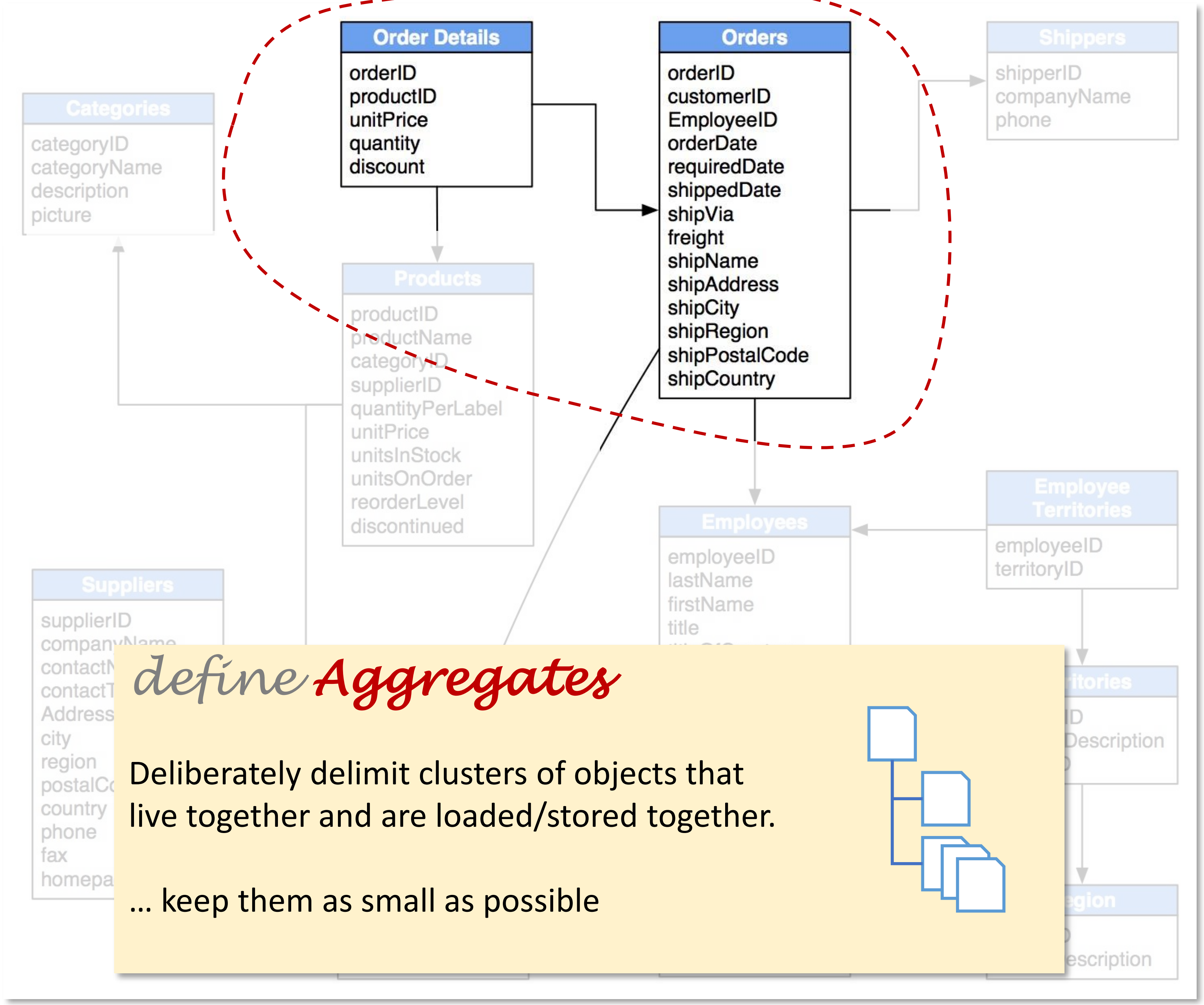
- Decoupled
- Testable
- Easy to know where functionality should live
- It worked fine initially

But … wait a minute …

Object graphs

*When loading an Order from DB ... **what else should I load** ?*

- All the relationships ?
- Some of them, and leave some unpopulated (`null`) ?
- Some of them, and use lazy-loading ?

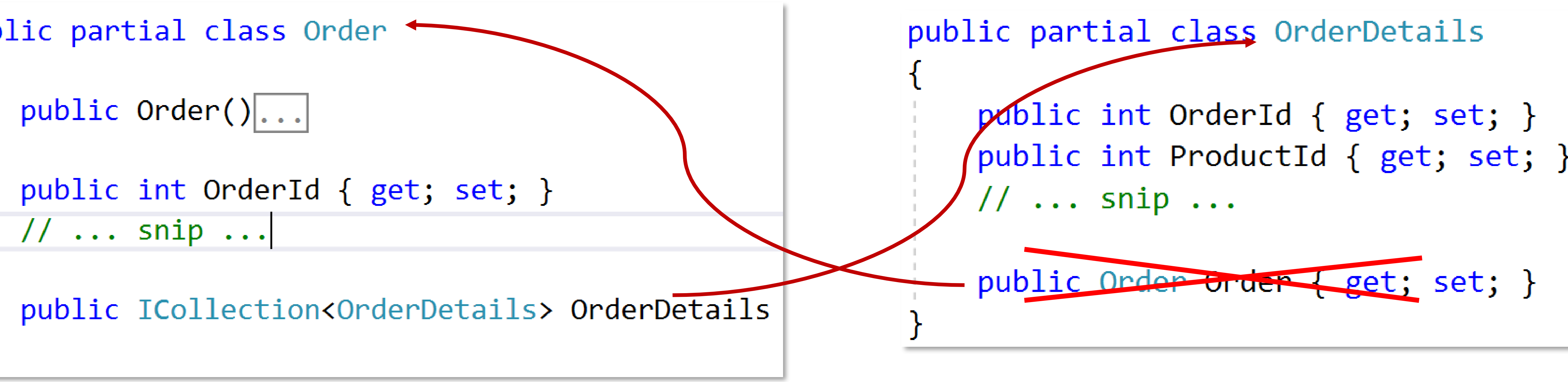


define ***Aggregates***

Deliberately delimit clusters of objects that live together and are loaded/stored together.

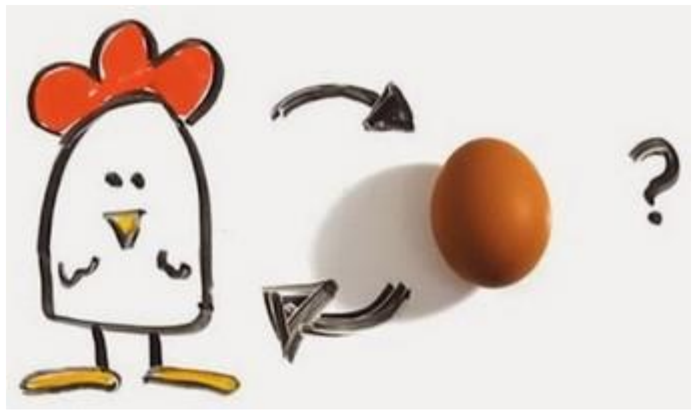
... keep them as small as possible

Object graphs



```
lic partial class Order

 public Order()...

 public int OrderId { get; set; }
 // ... snip ...

 public ICollection<OrderDetails> OrderDetails
```

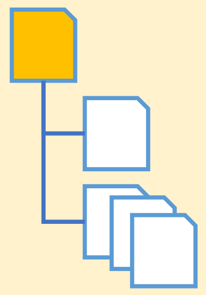
```
public partial class OrderDetails
{
    public int OrderId { get; set; }
    public int ProductId { get; set; }
    // ... snip ...

    ~~public Order Order { get; set; }~~
}
```



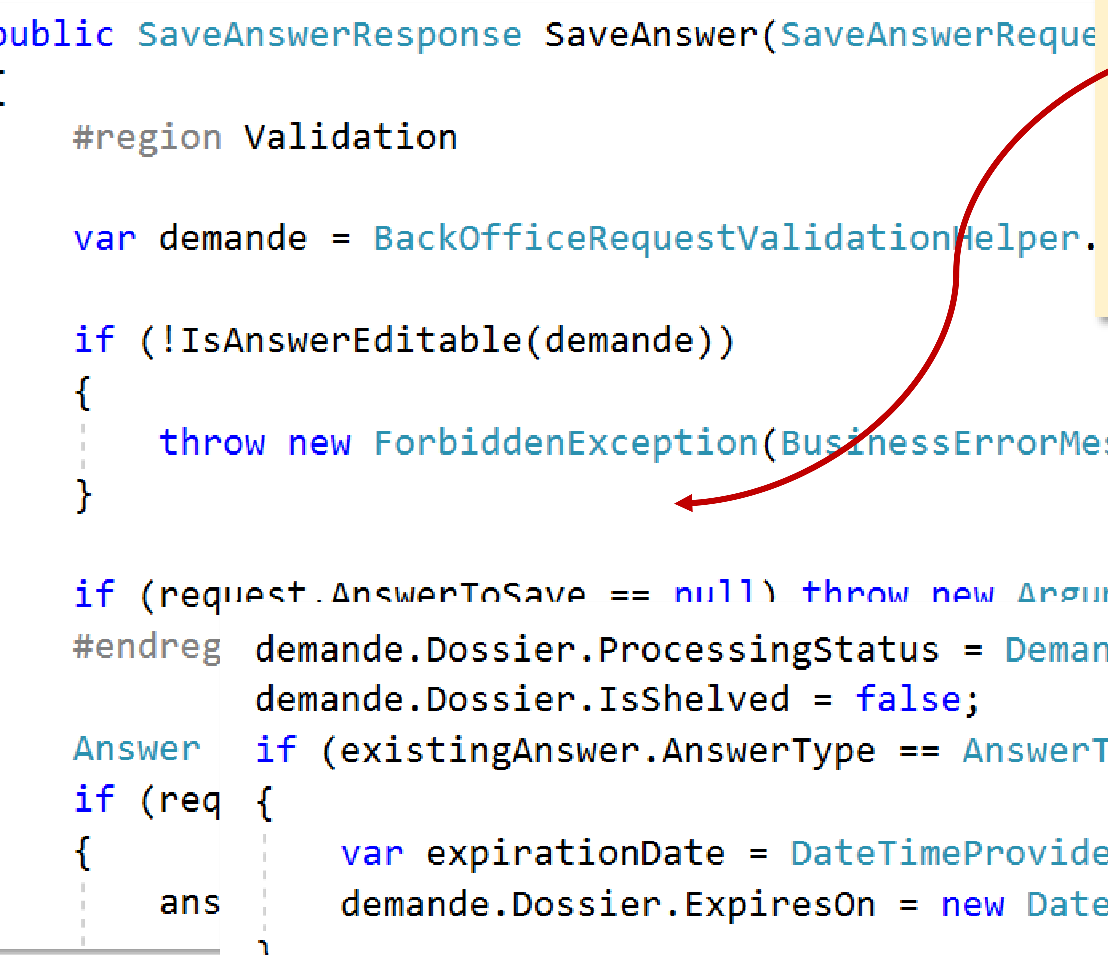
identify the ***Aggregate Root***

The unique entry point to the graph
Dependencies go only in one direction

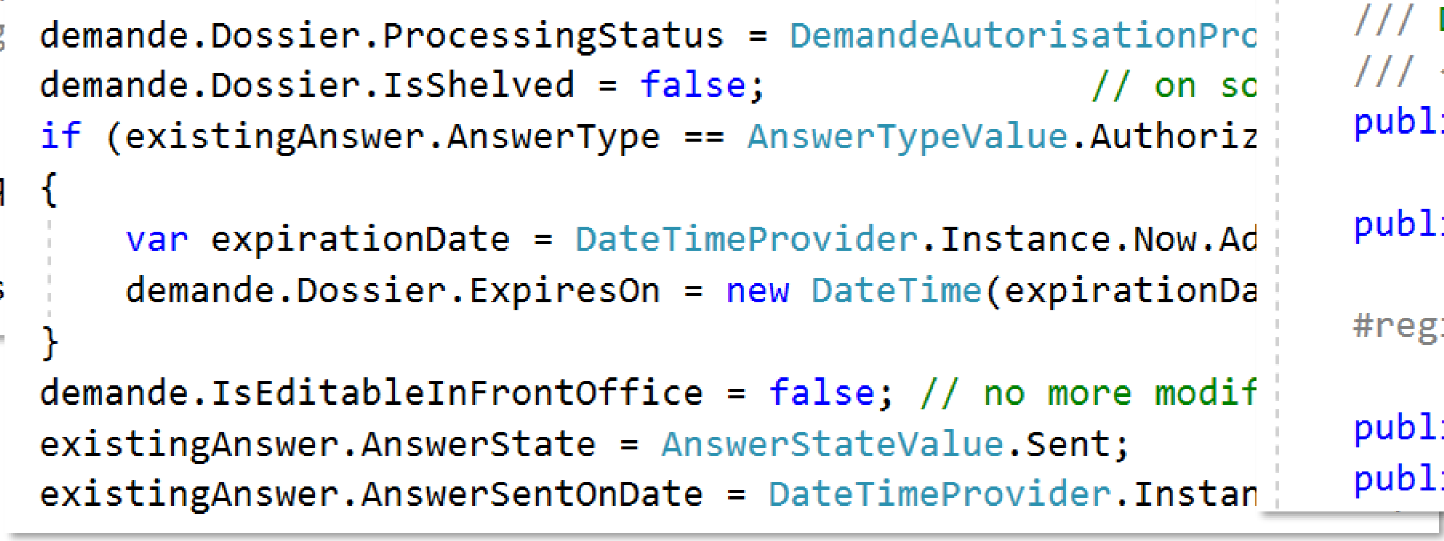
Repositories return Aggregates through their root



More smells …



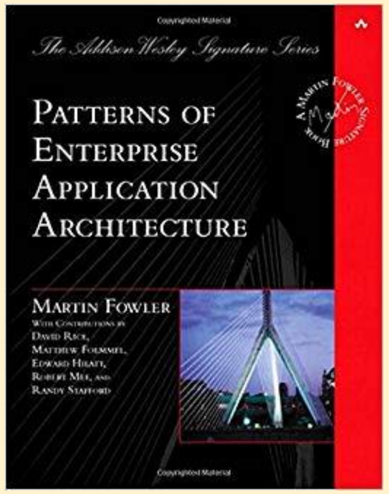
```
public SaveAnswerResponse SaveAnswer(SaveAnswerReque
{
    #region Validation

    var demande = BackOfficeRequestValidationHelper.

    if (!IsAnswerEditable(demande))
    {
        throw new ForbiddenException(BusinessErrorMessages.DemandeNot
    }

    if (request.AnswerToSave == null) throw new ArgumentNullException
    #endreg

    Answer
    if (req
    {
        ans
```

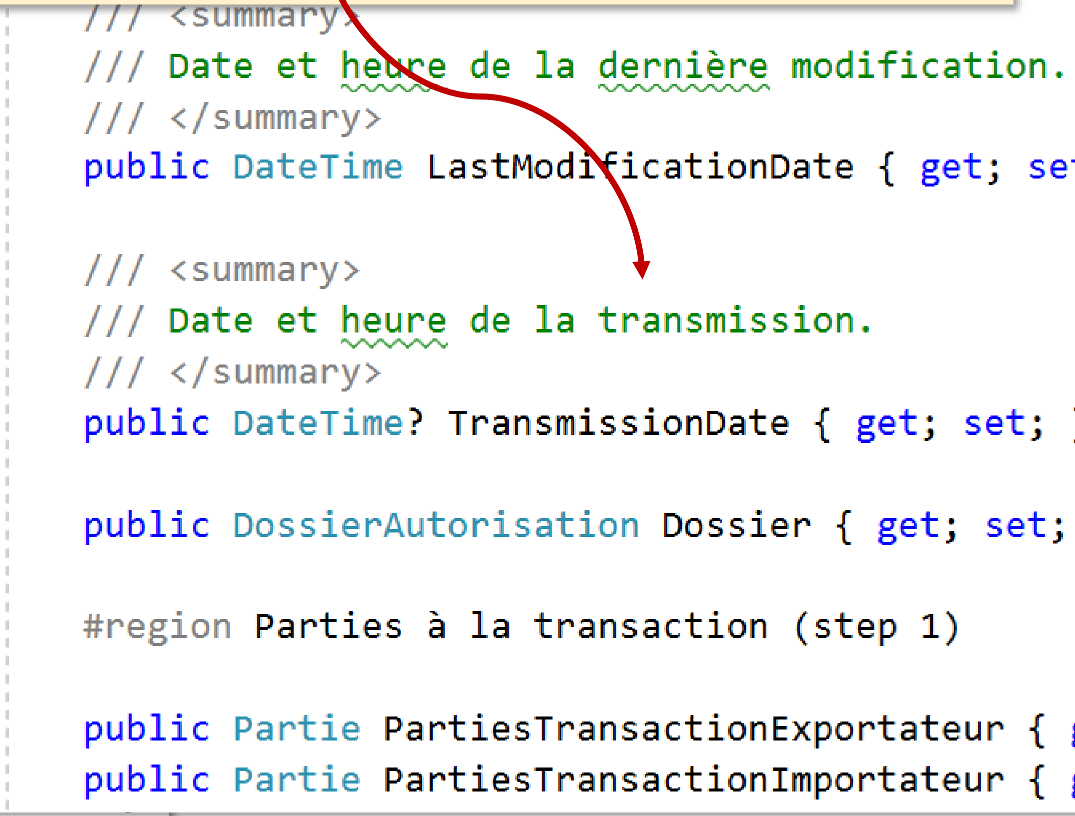


```
demande.Dossier.ProcessingStatus = DemandeAutorisationPro
demande.Dossier.IsShelved = false;              // on sc
if (existingAnswer.AnswerType == AnswerTypeValue.Authoriz
{
    var expirationDate = DateTimeProvider.Instance.Now.Ad
    demande.Dossier.ExpiresOn = new DateTime(expirationDa
}
demande.IsEditableInFrontOffice = false; // no more modif
existingAnswer.AnswerState = AnswerStateValue.Sent;
existingAnswer.AnswerSentOnDate = DateTimeProvider.Instan
```

Transaction Script

Anemic Domain Model


The Addison-Wesley Signature Series
Patterns of Enterprise Application Architecture
Martin Fowler
With Contributions by David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford



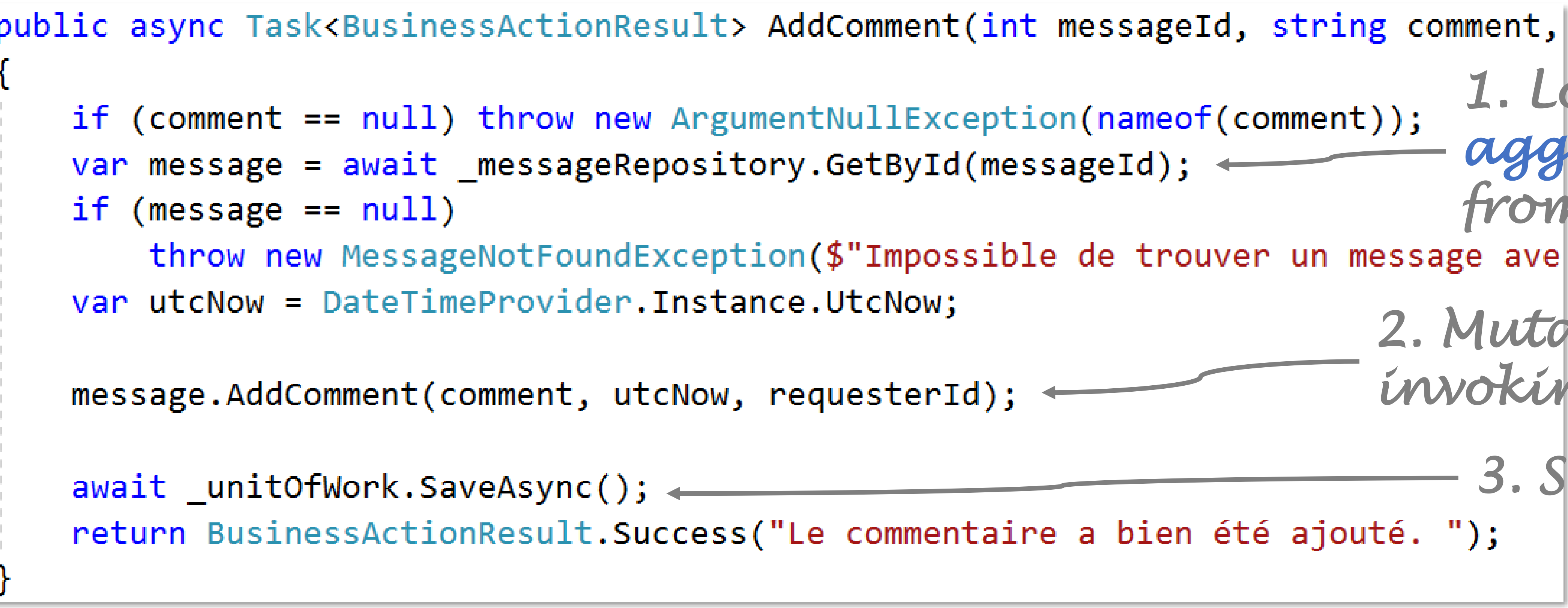
```
/// <summary>
/// Date et heure de la dernière modification.
/// </summary>
public DateTime LastModificationDate { get; se

/// <summary>
/// Date et heure de la transmission.
/// </summary>
public DateTime? TransmissionDate { get; set; }

public DossierAutorisation Dossier { get; set;

#region Parties à la transaction (step 1)

public Partie PartiesTransactionExportateur {
public Partie PartiesTransactionImportateur {
```

Fighting the Anemic Domain Model

- Do not expose setters
- Do expose only the Aggregate Root
- Do enforce invariants (guard clauses)
- Do mutations only through methods
- Use intent-revealing names (no `Update(…)` methods)

*Make **invalid** states **impossible** to represent*

Transaction Script -> Application Service



```
ublic async Task<BusinessActionResult> AddComment(int messageId, string comment,
{
    if (comment == null) throw new ArgumentNullException(nameof(comment));
    var message = await _messageRepository.GetById(messageId);
    if (message == null)
        throw new MessageNotFoundException($"Impossible de trouver un message ave
    var utcNow = DateTimeProvider.Instance.UtcNow;

    message.AddComment(comment, utcNow, requesterId);

    await _unitOfWork.SaveAsync();
    return BusinessActionResult.Success("Le commentaire a bien été ajouté. ");
}
```

1. Load *aggregate root* from repository

2. Mutate by invoking *methods*

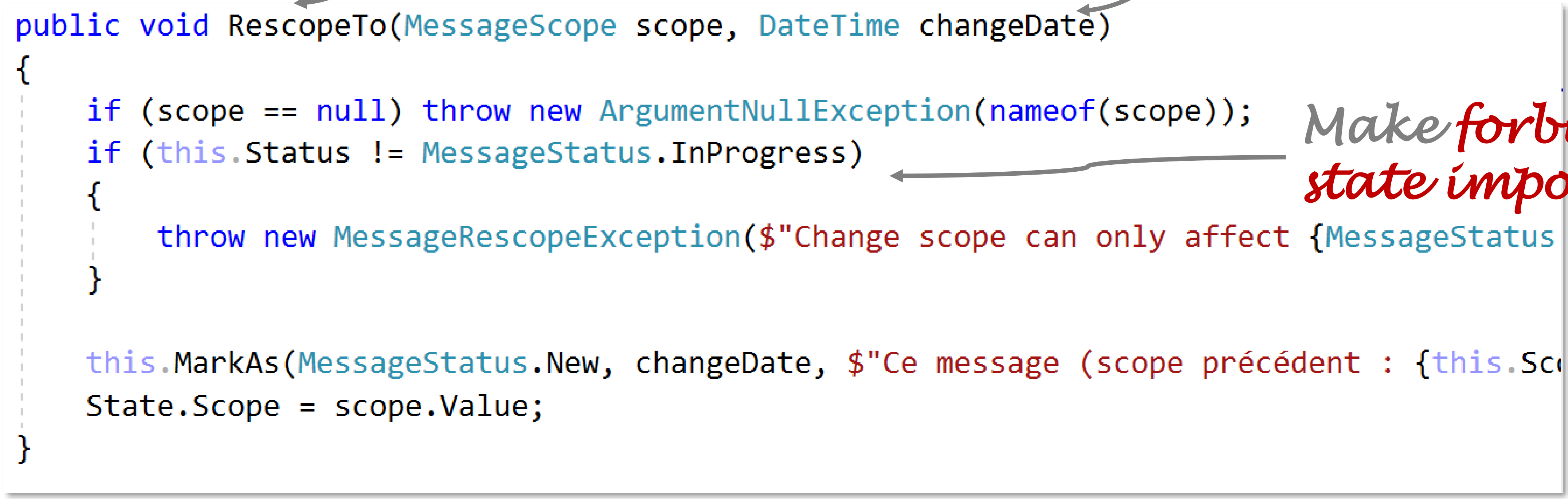
3. Save to DB

Methods on Domain Entities

Intent-revealing name

Avoid primitive types

Make forbidden state impossible

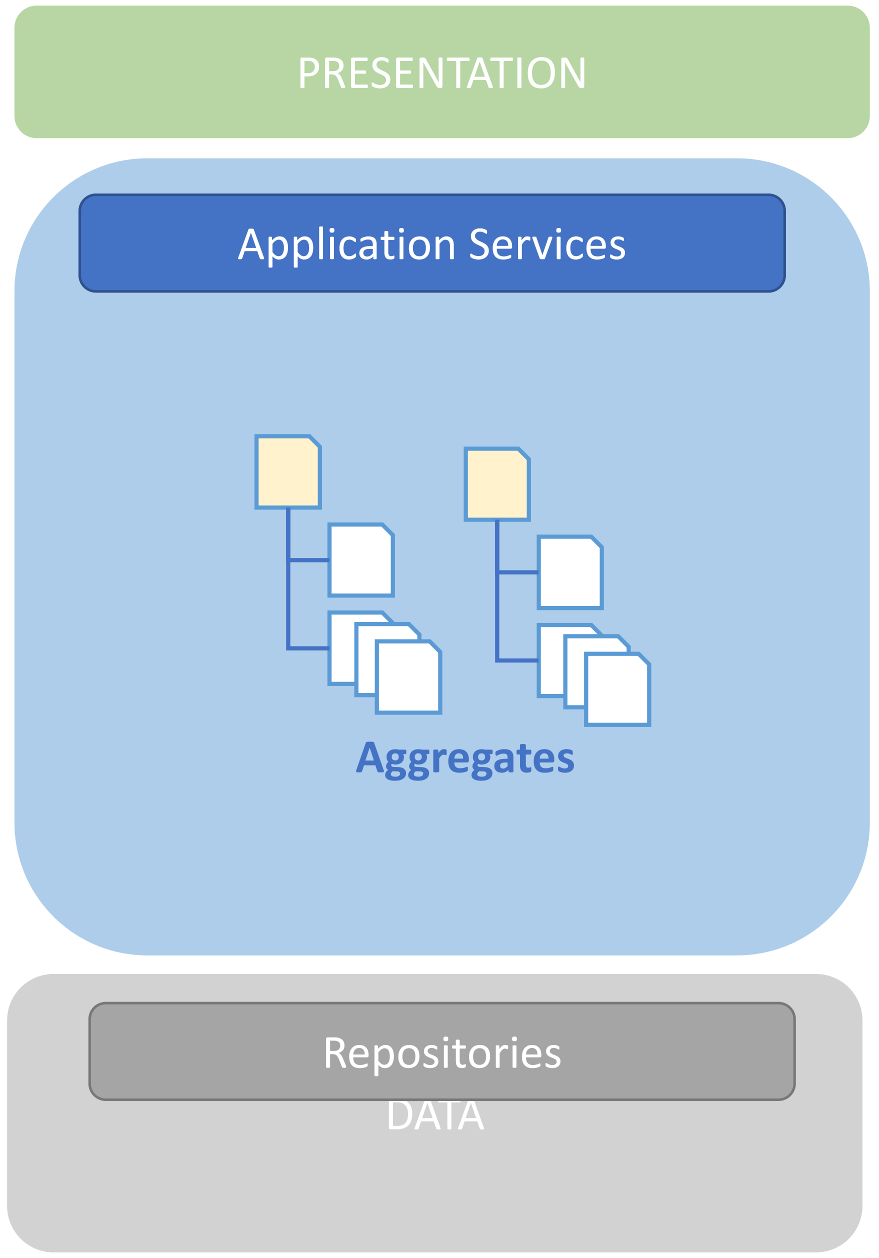


```
public void RescopeTo(MessageScope scope, DateTime changeDate)
{
    if (scope == null) throw new ArgumentNullException(nameof(scope));
    if (this.Status != MessageStatus.InProgress)
    {
        throw new MessageRescopeException($"Change scope can only affect {MessageStatus
    }

    this.MarkAs(MessageStatus.New, changeDate, $"Ce message (scope précédent : {this.Sc
    State.Scope = scope.Value;
}
```

Status

Even better

- Decoupled
- Testable
- Easy to know where functionality should live
- Explicit models and methods
- Clean encapsulation
- Meaningful names



What about views ?

We defined

- Small aggregates
- Repositories targeting only Aggregate Roots
- Transactional consistency
- Repositories hiding data-access

For views / reports we need

- JOINs across many tables
- Small ad-hoc queries on some tables
- No transactions
- Access to raw data / perf

*Different **needs** require different **tools***

Separating Reads and Writes

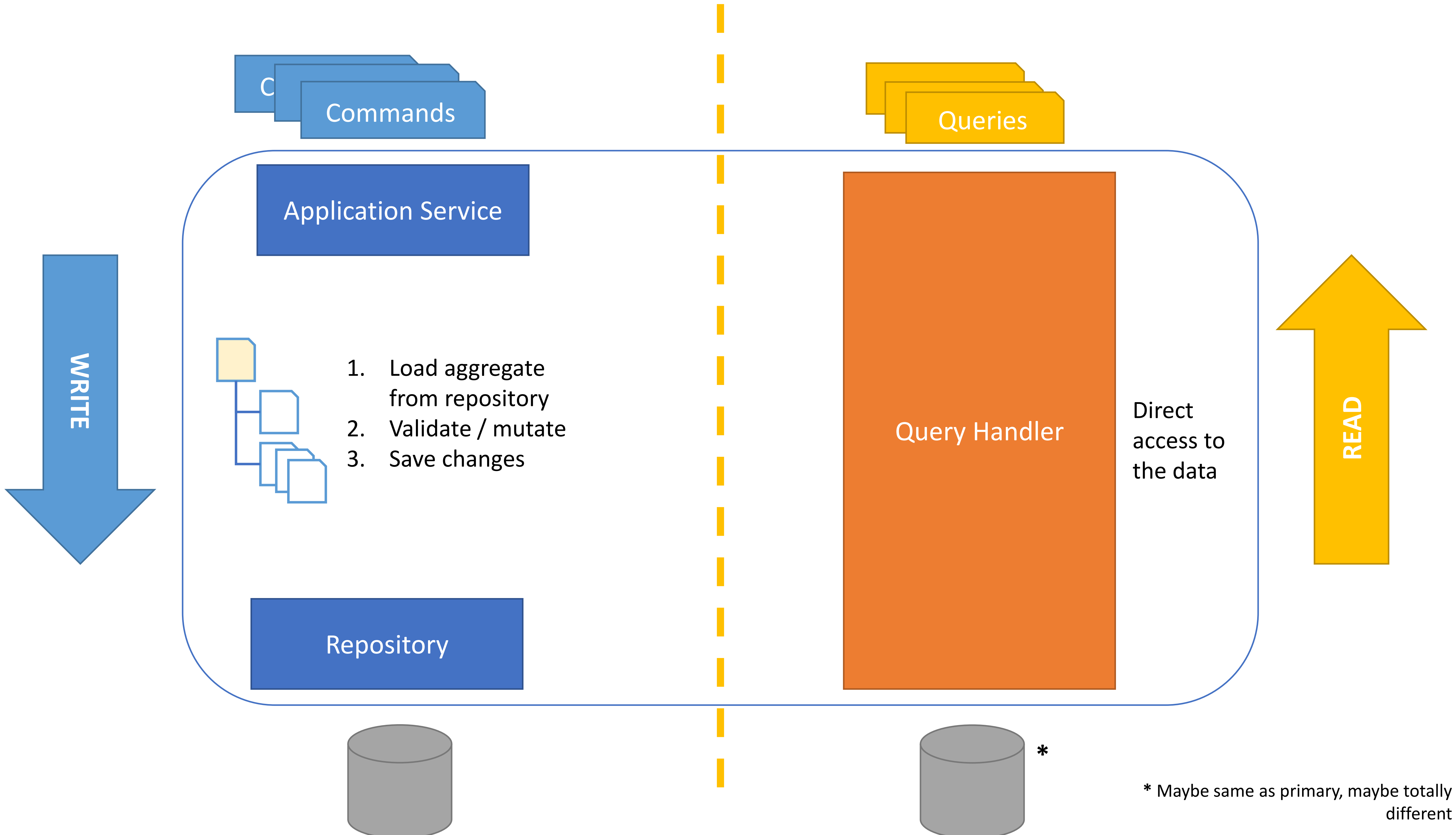
- Commands vs Queries
- Write Model vs Read Model

Aggregates

View-specific projections
As big or small as needed

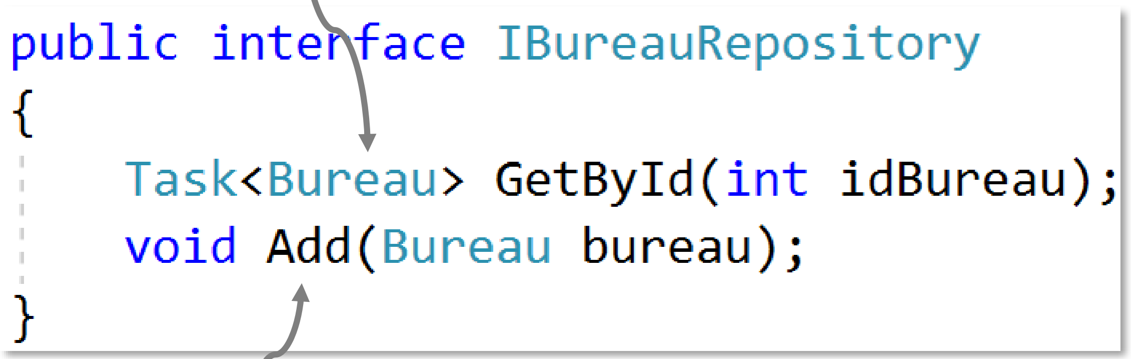
CQS (**C**ommand **Q**uery **S**eparation)

CQRS (**C**ommand **Q**uery **R**esponsibility **S**egregation)


Commands
Queries
Application Service
WRITE
1. Load aggregate from repository
2. Validate / mutate
3. Save changes
Query Handler
Direct access to the data
READ
Repository
*
* Maybe same as primary, maybe totally different

Write Model

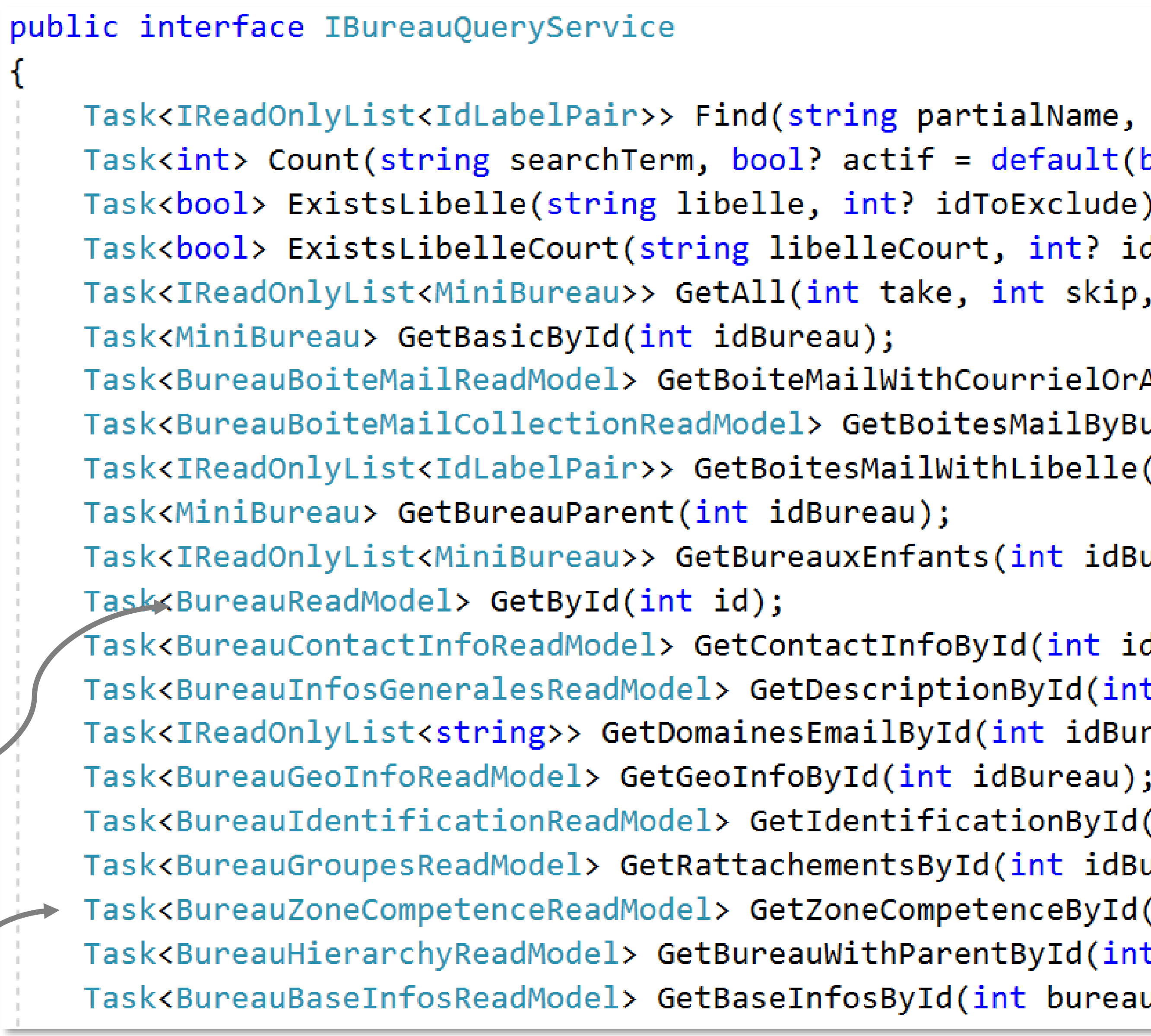
Expose only *Aggregates*



```
public interface IBureauRepository
{
    Task<Bureau> GetById(int idBureau);
    void Add(Bureau bureau);
}
```

Allow edits

Read Model



```
public interface IBureauQueryService
{
    Task<IReadOnlyList<IdLabelPair>> Find(string partialName,
    Task<int> Count(string searchTerm, bool? actif = default(b
    Task<bool> ExistsLibelle(string libelle, int? idToExclude)
    Task<bool> ExistsLibelleCourt(string libelleCourt, int? id
    Task<IReadOnlyList<MiniBureau>> GetAll(int take, int skip,
    Task<MiniBureau> GetBasicById(int idBureau);
    Task<BureauBoiteMailReadModel> GetBoiteMailWithCourrielOrA
    Task<BureauBoiteMailCollectionReadModel> GetBoitesMailByBu
    Task<IReadOnlyList<IdLabelPair>> GetBoitesMailWithLibelle(
    Task<MiniBureau> GetBureauParent(int idBureau);
    Task<IReadOnlyList<MiniBureau>> GetBureauxEnfants(int idBu
    Task<BureauReadModel> GetById(int id);
    Task<BureauContactInfoReadModel> GetContactInfoById(int id
    Task<BureauInfosGeneralesReadModel> GetDescriptionById(int
    Task<IReadOnlyList<string>> GetDomainesEmailById(int idBur
    Task<BureauGeoInfoReadModel> GetGeoInfoById(int idBureau);
    Task<BureauIdentificationReadModel> GetIdentificationById(
    Task<BureauGroupesReadModel> GetRattachementsById(int idBu
    Task<BureauZoneCompetenceReadModel> GetZoneCompetenceById(
    Task<BureauHierarchyReadModel> GetBureauWithParentById(int
    Task<BureauBaseInfosReadModel> GetBaseInfosById(int bureau
```

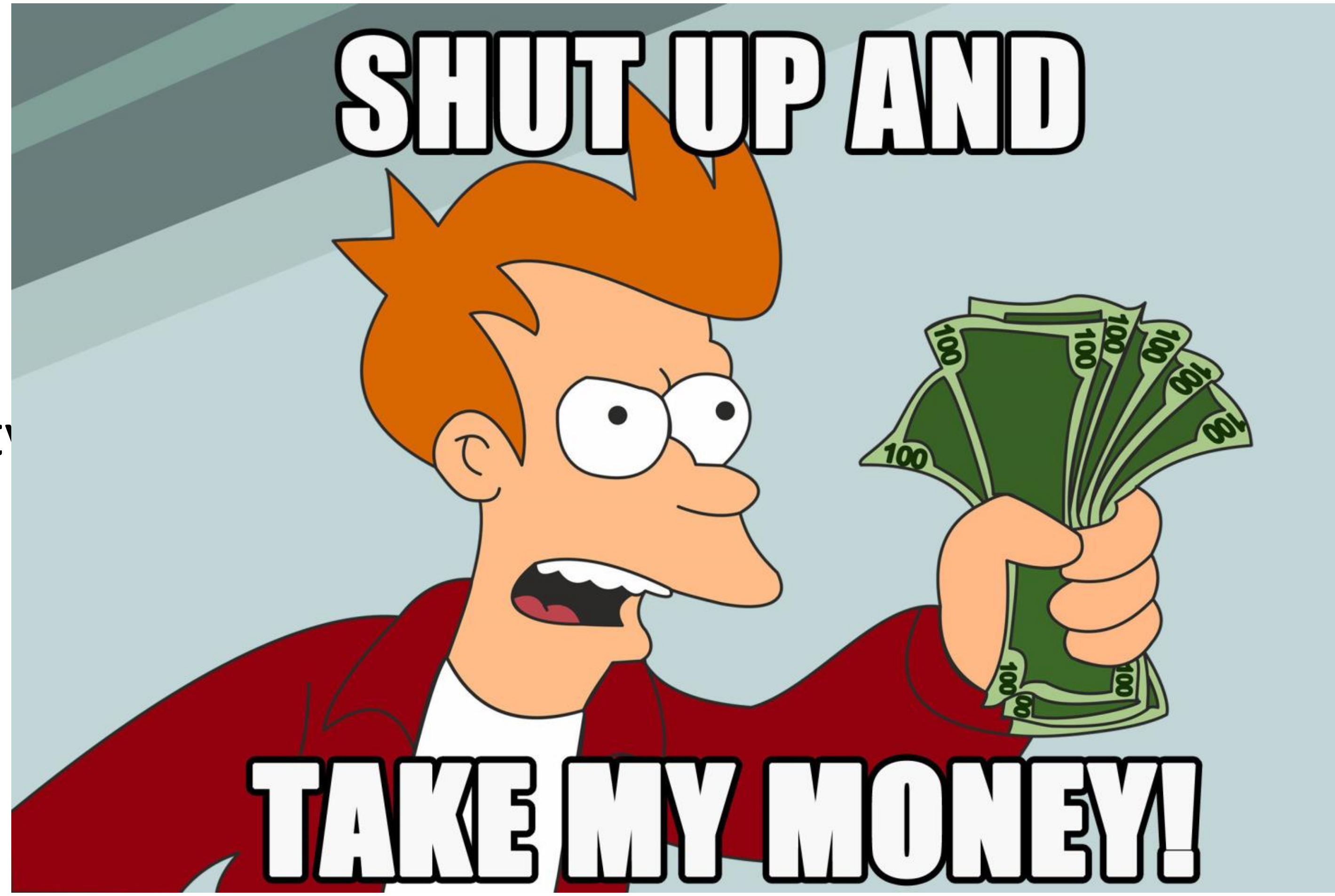
Expose *projections,* DTOs *tailored* for the *view*

Read-only

Status

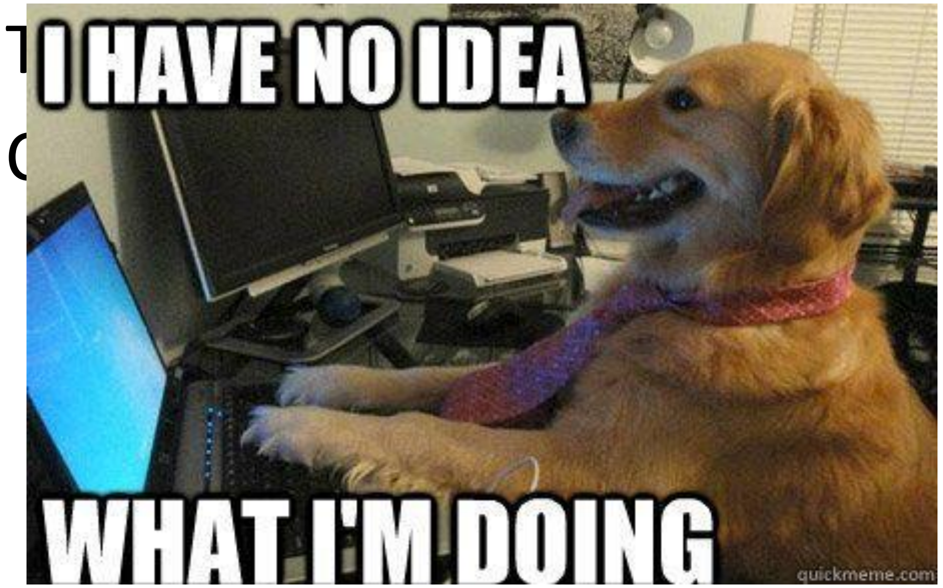
Good enough for now ☺

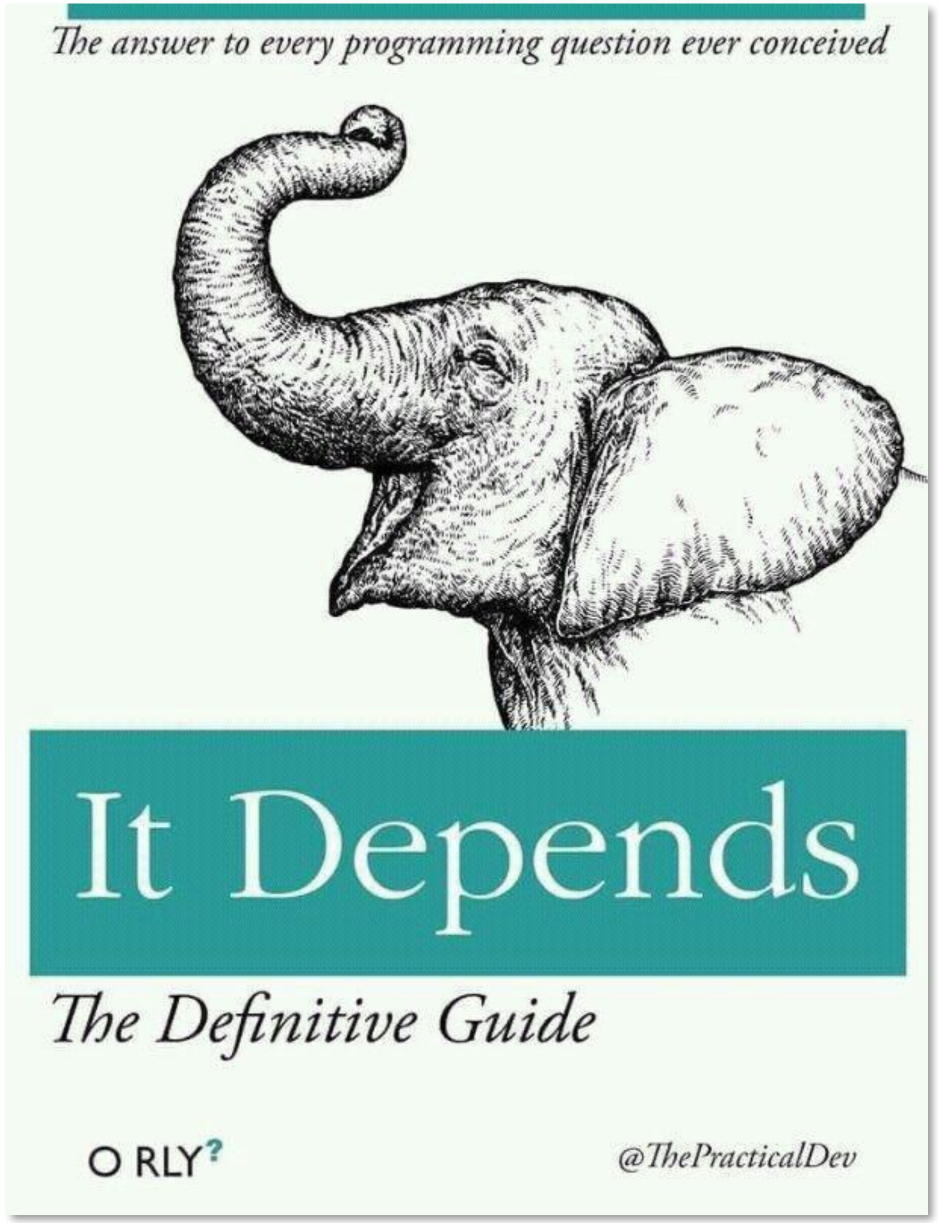
- Decoupled
- Testable
- Easy to know where functionalit should live
- Explicit models and methods
- Clean encapsulation
- Meaningful names
- Correct writes
- Fast Reads



Take aways

- Smoothly introduce DDD concepts
  - Failing is learning
  - Watch out for smells
  - Continuously improve
- T
- C

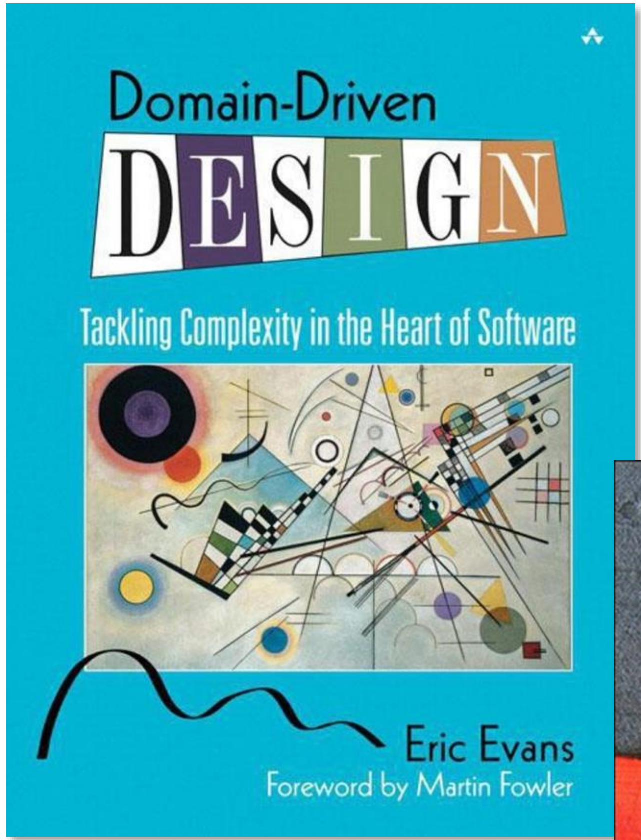


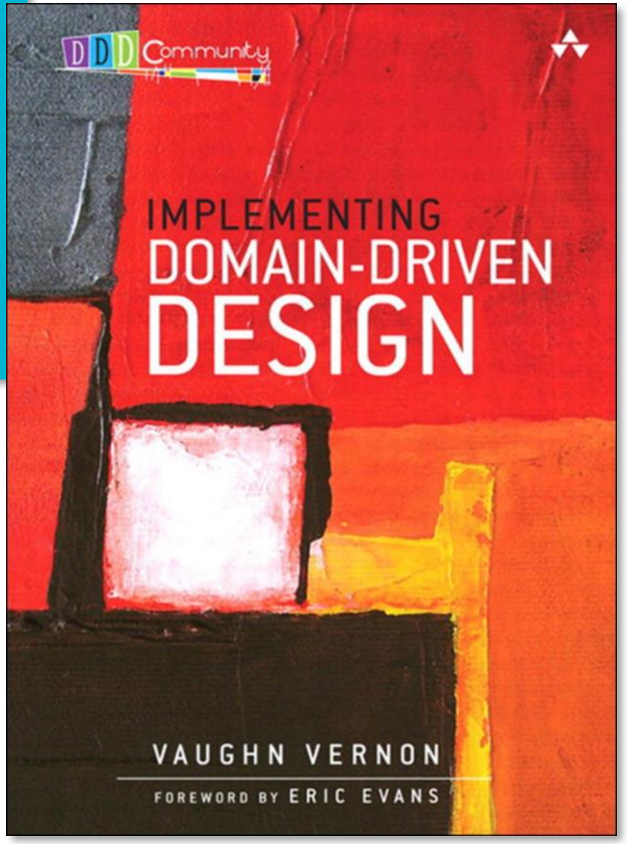


Going further

Some important concepts I left out :

- Value Objects
- Bounded Contexts
- Ubiquitous Language
- Domain Events
- Context Mapping
- …


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler



Thanks for listening !

Questions ?

Thibaud Desodt - @tsimbalar