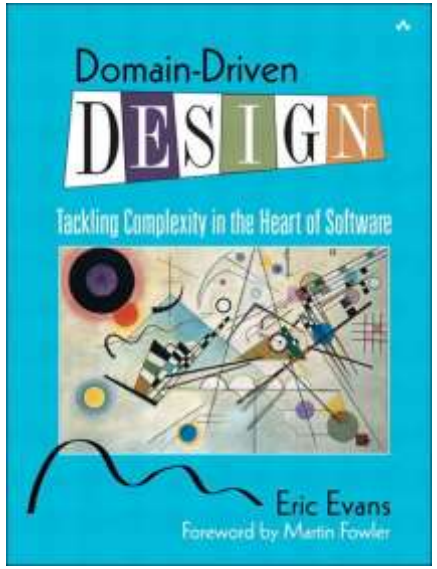
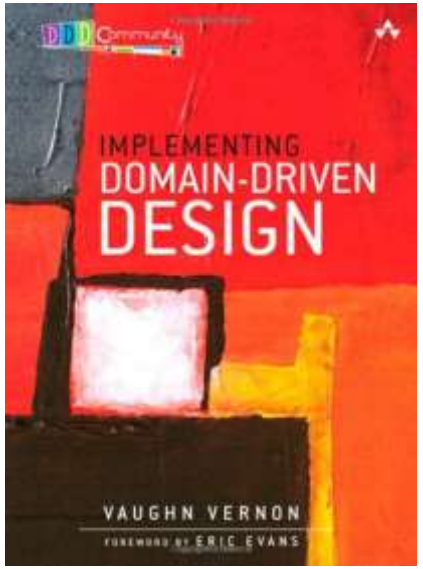




DomainDrivenDesign();

TACKLING COMPLEXITY IN THE HEART OF SOFTWARE

Problems of Enterprise Software

Maintenance, as software grows.

Mixing presentation, application and domain logics.

Mixing infrastructure and business code, makes hard to switch technology.

Testability

Before DDD

- Developers are such tool lovers and they believe that problems can solve only tools but re-considering of architecture
  - Let’s put Cassandra, Let’s put Messaging abstractions etc..
- DB oriented systems
- Tightly coupled services
- The hidden business inside of UtilityHelpers, Providers, BullShitManagers!
- Smart UI Antipattern
  - Put business logic to UserInterface
  - Create seperate user interface every application function
  - Duplicate business logic in Mobile, Mvc or inside other UI components
  - In Long-lived project lifetime, this is a nightmare

The Result is:

Architecture PER Developer!

What is DDD ?

NOT A TECHNOLOGY OR METHODOLOGY

- Set of principles and patterns focusing to design effort

Why DDD ?

Deep understanding the Domain(subject area), object model, interactions

Distilling your daily work language as **Ubiquitous Language**

Arrange and cover learning challenges across people

- Business Analyst, Domain Expert, Developer

Leads model driven design not Database!

- Creates a language
- Distilled by knowledge over the time

How can I decide to DDD?

	Competitive Advantage	Complexity
Accounting	S	L
Public Web	L	S
Claims	L	L
Sales	S	S

Overall

Ubiquitous Language

- Knowledge Crunching, Continuous Learning, Knowledge-Rich Design, Deep Models.
- One team, one language
- Code base should be based on that language

Bounded Context (Microservices?)

Layered Architecture

- Domain, Application, Presentation, Infrastructure, Distributes Services

Layers

Domain Layer

- Entities, Aggregates, Aggregate Roots, Value Objects, Repositories, Domain Services, Domain Events, Policies

Application Layer

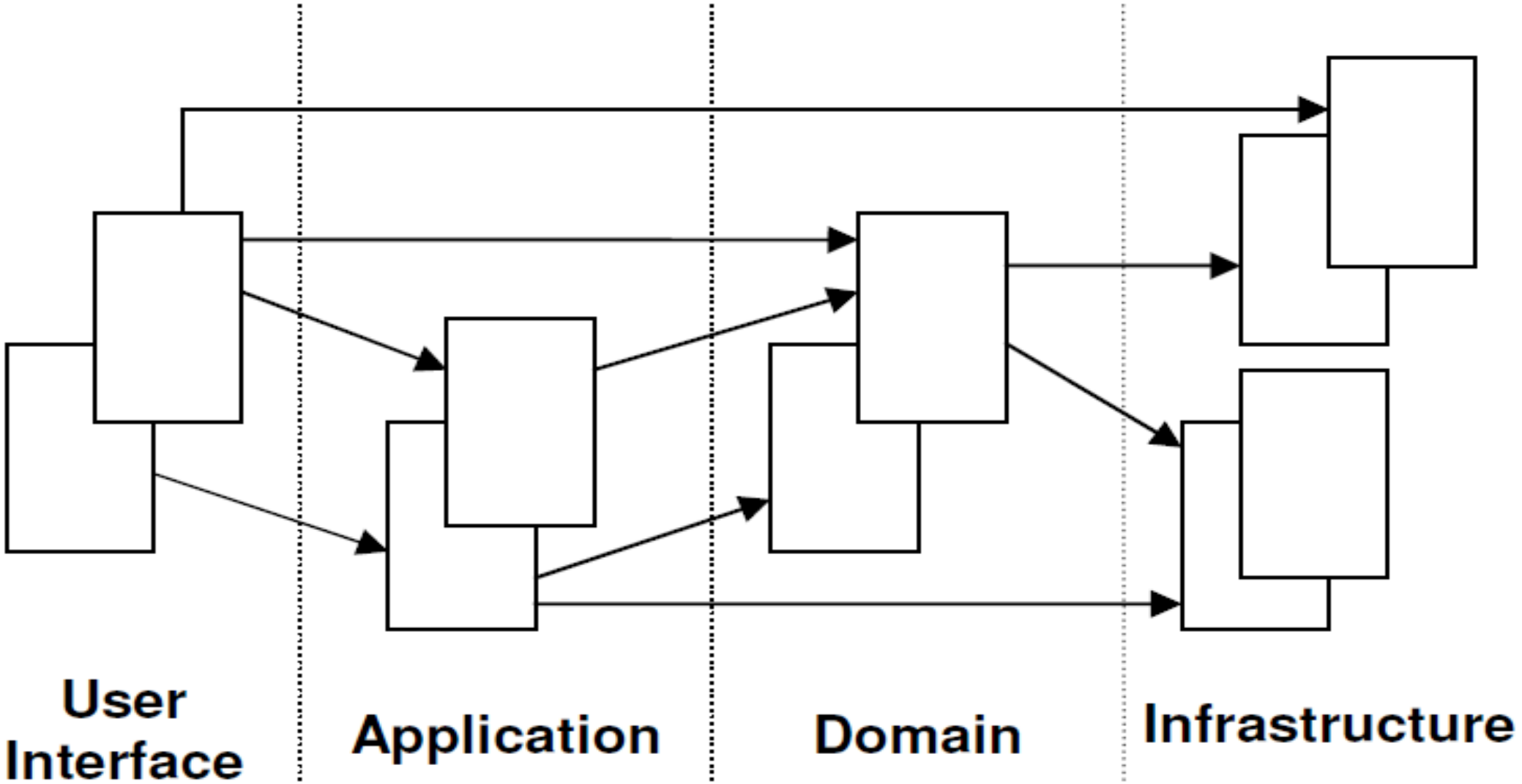
- Application Services, Data Transfer Objects (and their Validation), Unit Of Work, Authorization, Auditing…

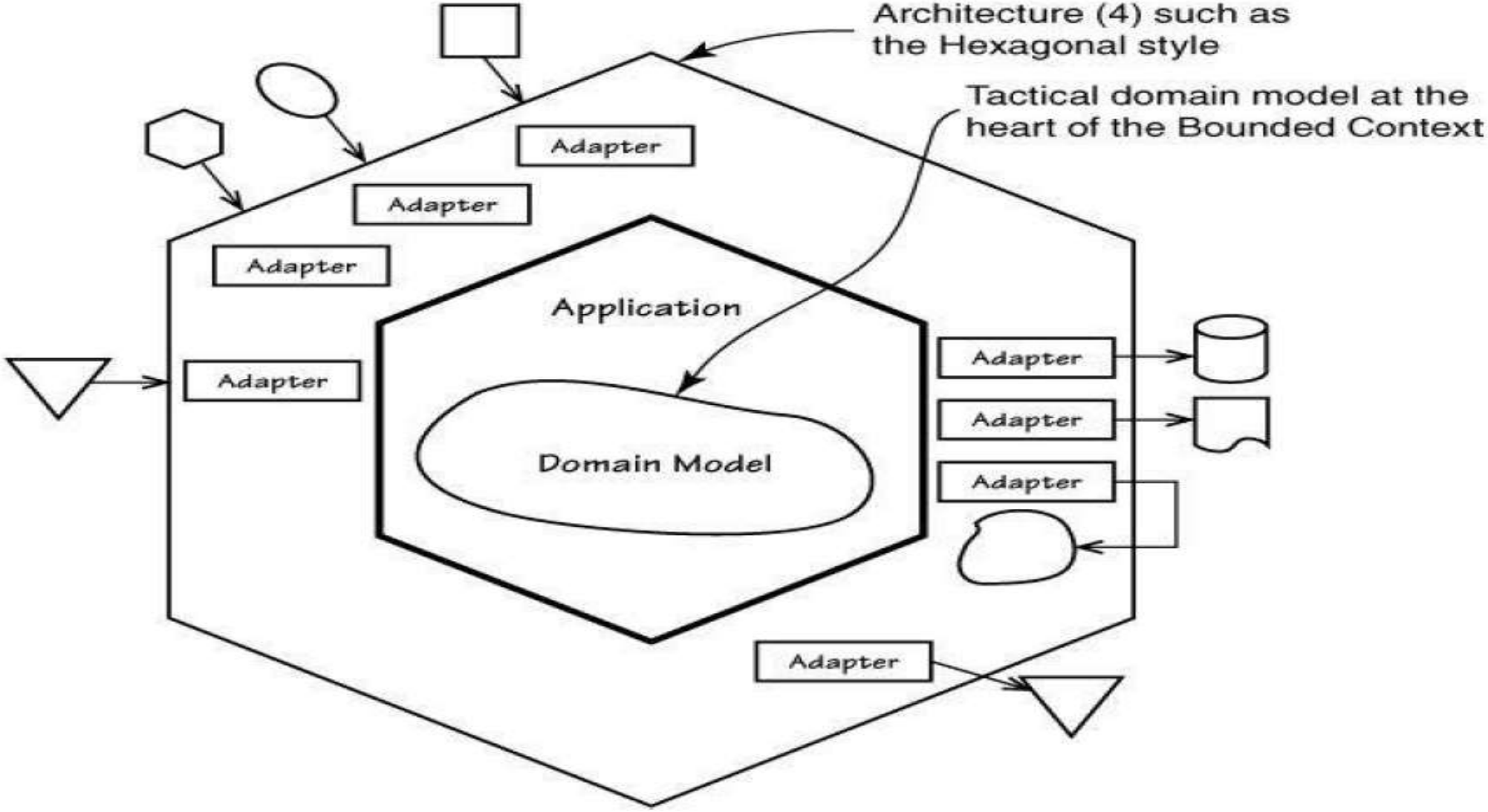
Infrastructure Layer

- Persistence, Sending Emails, Dependency Injection, Logging, Caching implementations…

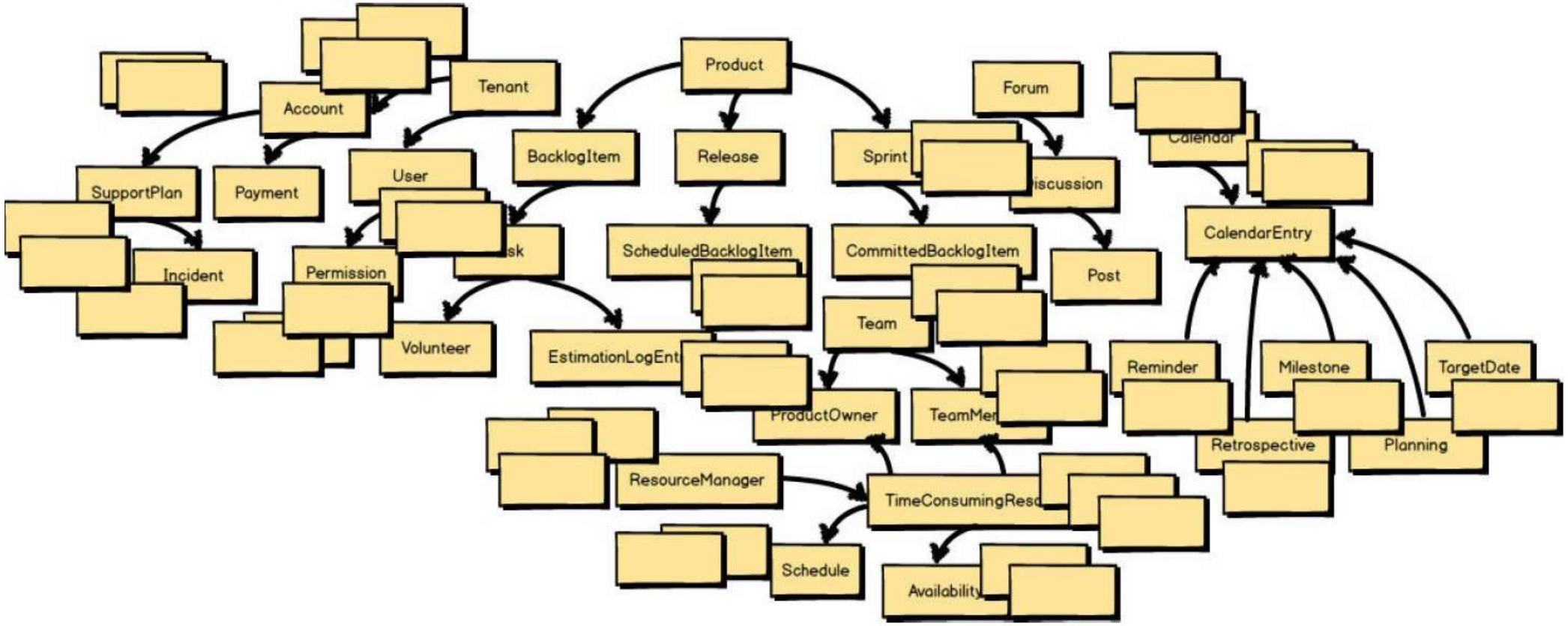
Presentation Layer

- Web, Mobile… specific technologies and tools

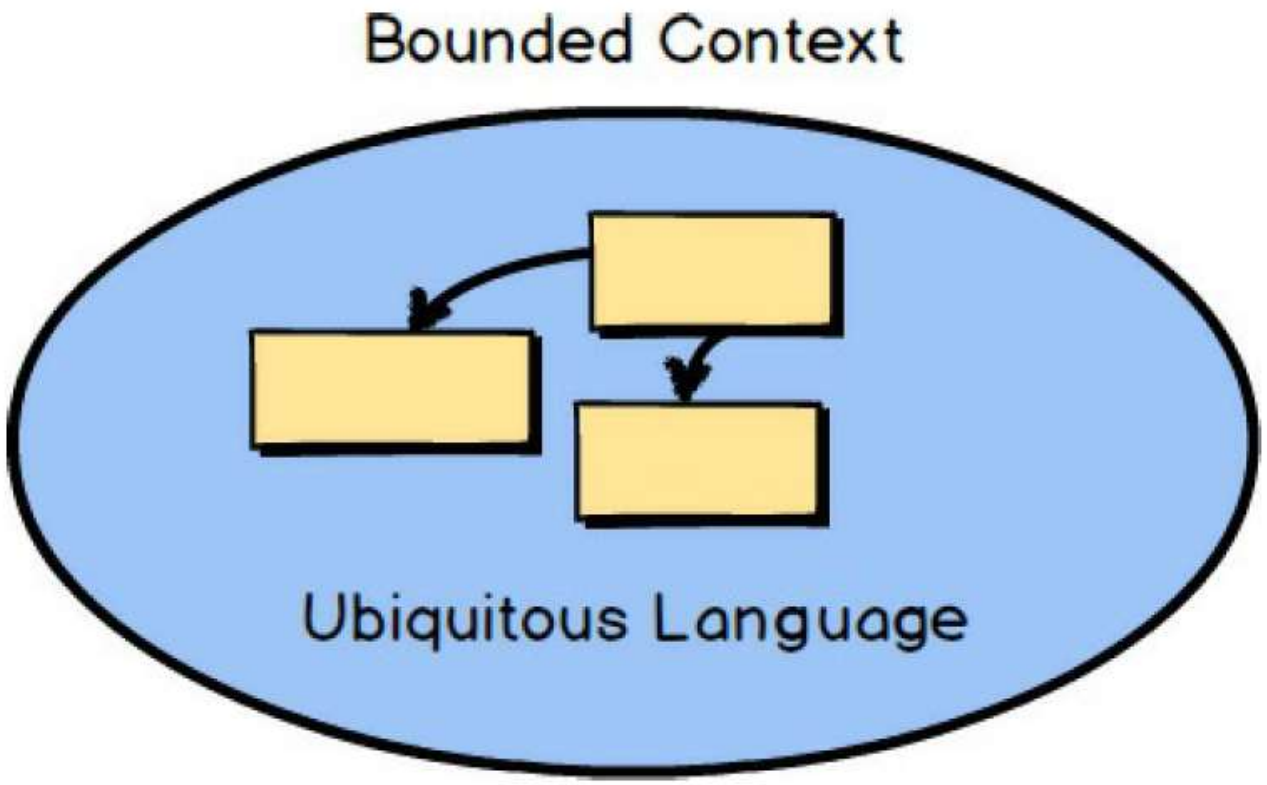

User Interface
Application
Domain
Infrastructure


Architecture (4) such as
the Hexagonal style
Tactical domain model at the
heart of the Bounded Context
Adapter
Adapter
Adapter
Application
Adapter
Domain Model
Adapter
Adapter
Adapter
Adapter

Non DDD Applied

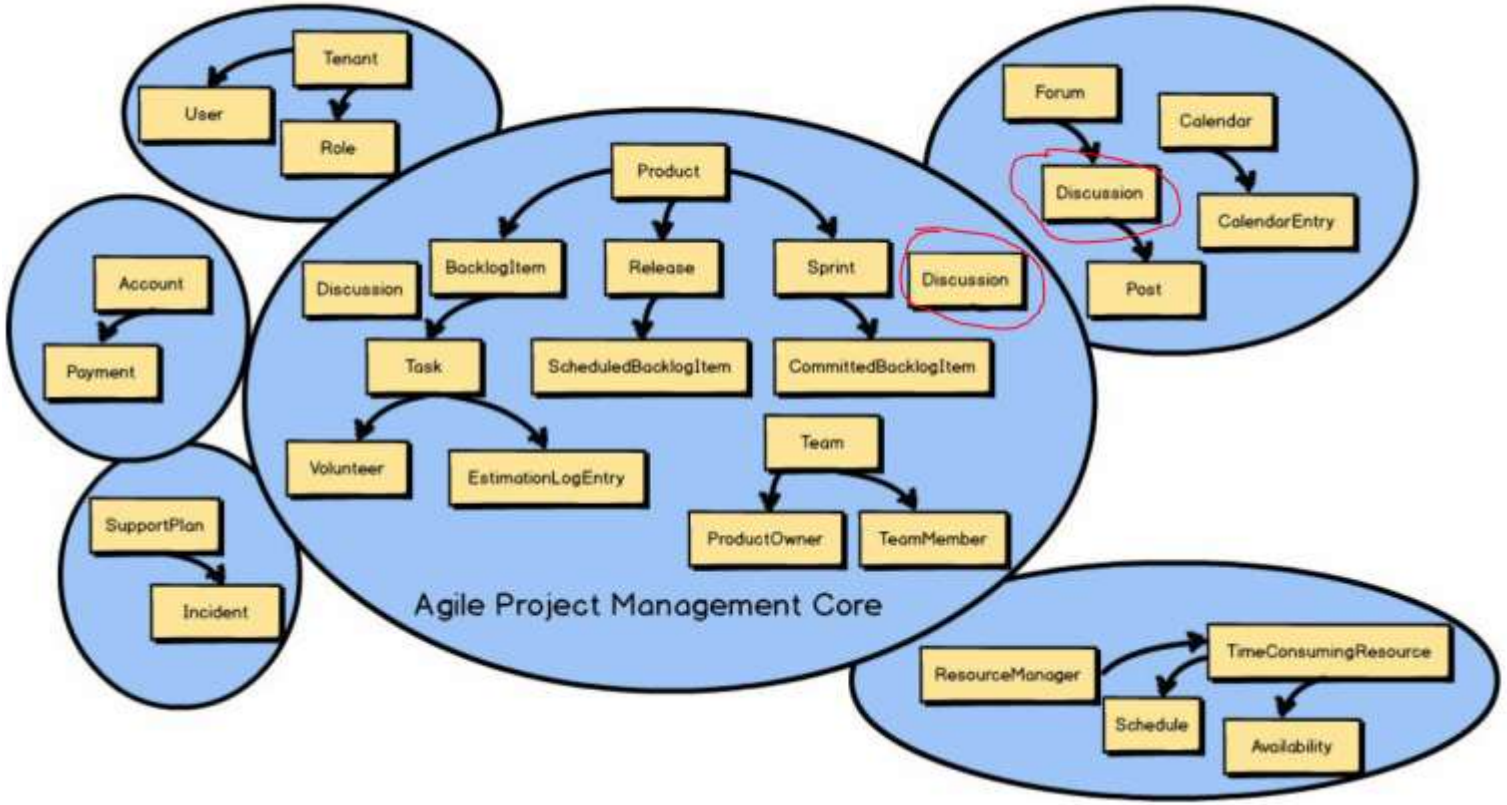


Bounded Contexts: Linguistic Boundary



- Give meaning to your software model
- Put your all relational models in same linguistic boundary
- Define a ubiquitous language

DDD Applied


Tenant
User
Role
Account
Payment
SupportPlan
Incident
Product
Discussion
BacklogItem
Release
Sprint
Discussion
Task
ScheduledBacklogItem
CommittedBacklogItem
Volunteer
EstimationLogEntry
Team
ProductOwner
TeamMember
Agile Project Management Core
Forum
Calendar
Discussion
CalendarEntry
Post
ResourceManager
TimeConsumingResource
Schedule
Availability

Aggregate Roots

- Handles the business of Entities and invariants
- Core parts of your BoundedContext
- Aggregate Roots should/can know each other **only and only through their Id's**.
- Manipulate **one** AggreateRoot in one **UnitOfWork/Transaction** Scope
- If you need to talk other Aggregate Root then use Messaging/Eventing system. Otherwise if your coupling needs to put that relation under one Application Service method/Uow Scope **then consider your modelling→** Because Aggregate Roots should be eventually consistent parts of your domain.
- Otherwise your modelling is wrong.
- If you smelled your modelling is wrong, then Re-Think.
- Aggregate Roots & Entities must be persistence agnostic!

Aggregate Roots

- Aggregate Roots can consists more than one Entities and ValueObjects

```
26 references
public class Product : AggregateRoot
{
    private const int MaxBarcodeLength = 40;

    1 reference
    private Product()
    {
        ProductImages = new List<ProductImage>();
    }

    1 reference
    public virtual string Name { get; private set; }

    3 references
    public virtual string Barcode { get; private set; }

    1 reference
    public virtual string Code { get; private set; }

    3 references
    public virtual ICollection<ProductImage> ProductImages { get; private set; }

    0 references
    public virtual Price Price { get; private set; }
```

Base class for Events & Id

Value Object Of AR

Entity collections of AR

```
1 reference
public static Product Create(string name, string code, string barcode)
{
    if (barcode.Length > MaxBarcodeLength)
    {
        throw new BarcodeExceedsMaximumLengthException($"Product creation for {barcode} exceeds maximum length of {MaxBarcodeLength}");
    }

    var product = new Product();
    product.RaiseEvent(new ProductCreatedEvent(barcode, code, name));
    return product;
}

0 references
internal void Apply(ProductCreatedEvent @event)
{
    Name = @event.Name;
    Barcode = @event.Barcode;
    Code = @event.Code;
}
```

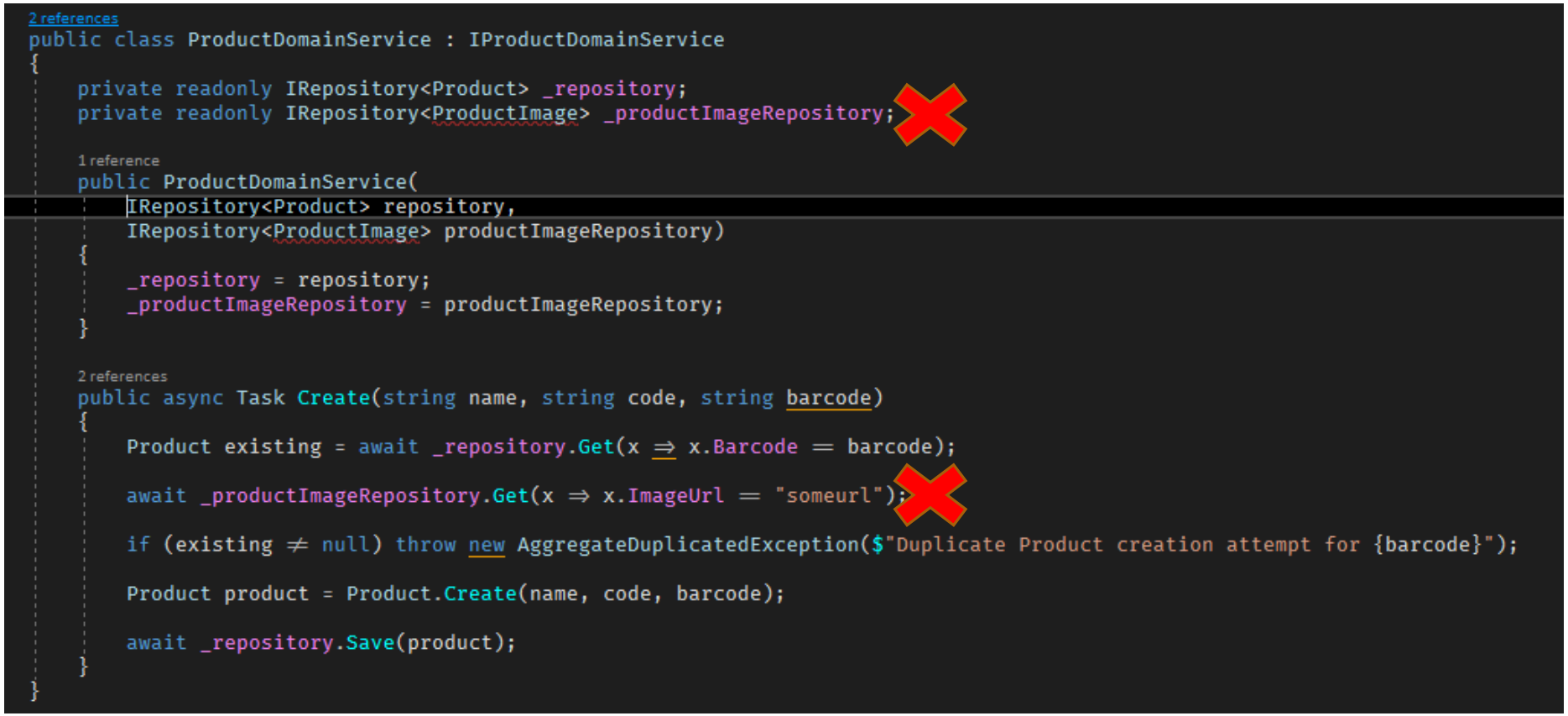
```
0 references
public void AddImage(string url, int order)
{
    if (ProductImages.Any(x => x.ShowOrder == order)) { throw new BusinessException($"Image order already defined for Product:{Id}"); }

    RaiseEvent(
        ProductImage.ImageAddedToProduct(this, url, order)
    );
}

0 references
internal void Apply(ImageAddedEvent @event)
{
    var image = new ProductImage();
    image.Route(@event);
    ProductImages.Add(image);
}
```

Aggregate Root Mistakes: Referring

```
public class Product : AggregateRoot
{
    private const int MaxBarcodeLength = 40;

    1 reference
    private Product()
    {
        ProductImages = new List<ProductImage>();
    }

    1 reference
    public virtual string Name { get; private set; }

    3 references
    public virtual string Barcode { get; private set; }

    1 reference
    public virtual string Code { get; private set; }

    0 references
    public virtual Order Order { get; private set; }

    3 references
    public virtual ICollection<ProductImage> ProductImages { get; protected set; }
```

```
public virtual int OrderId { get; private set; }
```

AR Mistakes: Entity Repo Injection



```
2 references
public class ProductDomainService : IProductDomainService
{
    private readonly IRepository<Product> _repository;
    private readonly IRepository<ProductImage> _productImageRepository;

    1 reference
    public ProductDomainService(
        IRepository<Product> repository,
        IRepository<ProductImage> productImageRepository)
    {
        _repository = repository;
        _productImageRepository = productImageRepository;
    }

    2 references
    public async Task Create(string name, string code, string barcode)
    {
        Product existing = await _repository.Get(x ⇒ x.Barcode == barcode);

        await _productImageRepository.Get(x ⇒ x.ImageUrl == "someurl");

        if (existing ≠ null) throw new AggregateDuplicatedException($"Duplicate Product creation attempt for {barcode}");

        Product product = Product.Create(name, code, barcode);

        await _repository.Save(product);
    }
}
```

Entities

- Have an identity which remains the same throughout the states of the software
- Components of Aggregate Roots
- Does not mean anything without its Aggregate Roots
- AggregateRoot manages the Entities and AR is responsible for validate it.
- Mutable(Attributes can change)
- Should have business logic
- Should be persistence agnostic

Sample Entity

Base class for Entity Behaviour with Id

When AR creates an Entity

Aggregate Root reference

```
5 references
public class ProductImage : Entity
{
    1 reference
    public ProductImage()
    {
        Register<ImageAddedEvent>(@event =>
        {
            Product = @event.Product;
            ImageUrl = @event.Url;
            ShowOrder = @event.Order;
        });
    }

    1 reference
    public virtual Product Product { get; private set; }

    1 reference
    public virtual string ImageUrl { get; private set; }

    2 references
    public virtual int ShowOrder { get; private set; }

    1 reference
    public static ImageAddedEvent ImageAddedToProduct(Product product, string url, int order)
    {
        if (string.IsNullOrEmpty(url))
        {
            throw new ArgumentNullException("url");
        }

        if (order <= 0)
        {
            throw new ArgumentException("order cannot be 0 or less");
        }

        return new ImageAddedEvent(product, url, order);
    }
}
```

Value Objects

- Immutable objects(Attributes can’t change, only replace whole)
- Have **no** identities
- Only purpose of describing some attributes of Aggregate Roots

```
3 references
public class Price : ValueObject<Price>
{
    public decimal Amount;
    public string Currency;

    0 references
    public Price(decimal amount, string currency)
    {
        Amount = amount;
        Currency = currency;
    }
}
```

Application Services

- Use Cases/Capabilities of your Bounded Context
- Manages the Transaction (Opens UoW)
- Knows Infra(EmailSender, Queue, Authorization etc...)
- D**oes not contain any business logic, does not hold any state!**
- Coordinates commands and behaviours of DomainServices.
- **Does not return never ever** Domain objects to outside world! Only returns Data Transfer Objects(DTO)

```
2 references
public class ProductAppService : IProductAppService
{
    private readonly IAuthorizationService _authorizationService;
    private readonly IProductDomainService _productDomainService;
    private readonly IUnitOfWorkManager _unitOfWorkManager;
    private readonly IMailSender _mailSender;
    private readonly ISession _session;
    private readonly IRealtimeNotifier _realtimeNotifier;

    1 reference
    public ProductAppService(
        IProductDomainService productDomainService,
        IAuthorizationService authorizationService,
        IUnitOfWorkManager unitOfWorkManager,
        IMailSender mailSender,
        ISession session,
        IRealtimeNotifier realtimeNotifier)
    {
        _productDomainService = productDomainService;
        _authorizationService = authorizationService;
        _unitOfWorkManager = unitOfWorkManager;
        _mailSender = mailSender;
        _session = session;
        _realtimeNotifier = realtimeNotifier;
    }

    2 references
    public async Task Handle(CreateProductCommand message)
    {
        _authorizationService.CheckPermission("User.CreateProduct");

        message.Validate();

        using (IUnitOfWork unitOfWork = _unitOfWorkManager.Begin(IsolationLevel.ReadCommitted))
        {
            await _productDomainService.Create(message.Name, message.Code, message.Barcode);

            await unitOfWork.Complete();
        }

        await _mailSender.Send($"Hello {_session.Username}, product is created for you.", "Product Creation", _session.Email);

        await _realtimeNotifier.Notify("ProductCreated", message.ToString());
    }
}
```

No DTO passing, just primitives or Domain Objects allowed

Transaction management

Domain Interaction

Infra & Messaging & Mailing

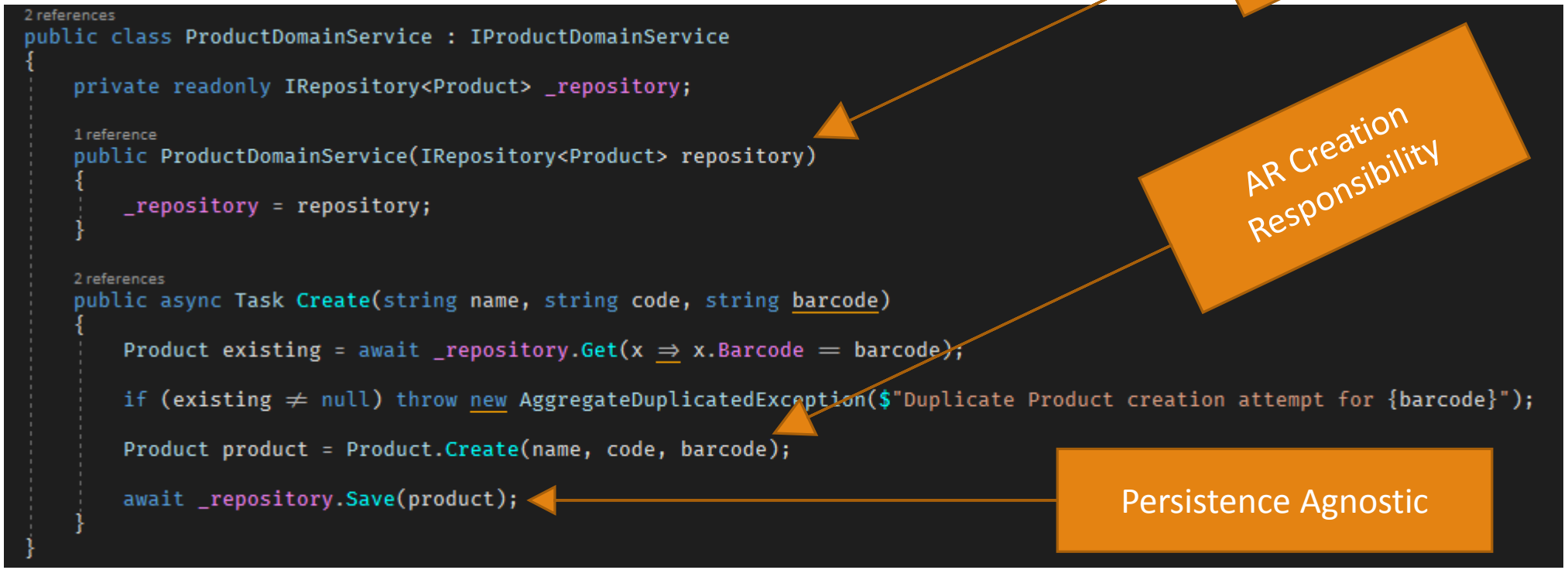
ApplicationService Mistakes

- Leaked business logic from the DomainService
- Utility, Helper, Provider driven coding intention
- Over abstraction, Over Reflection and other bullshits...

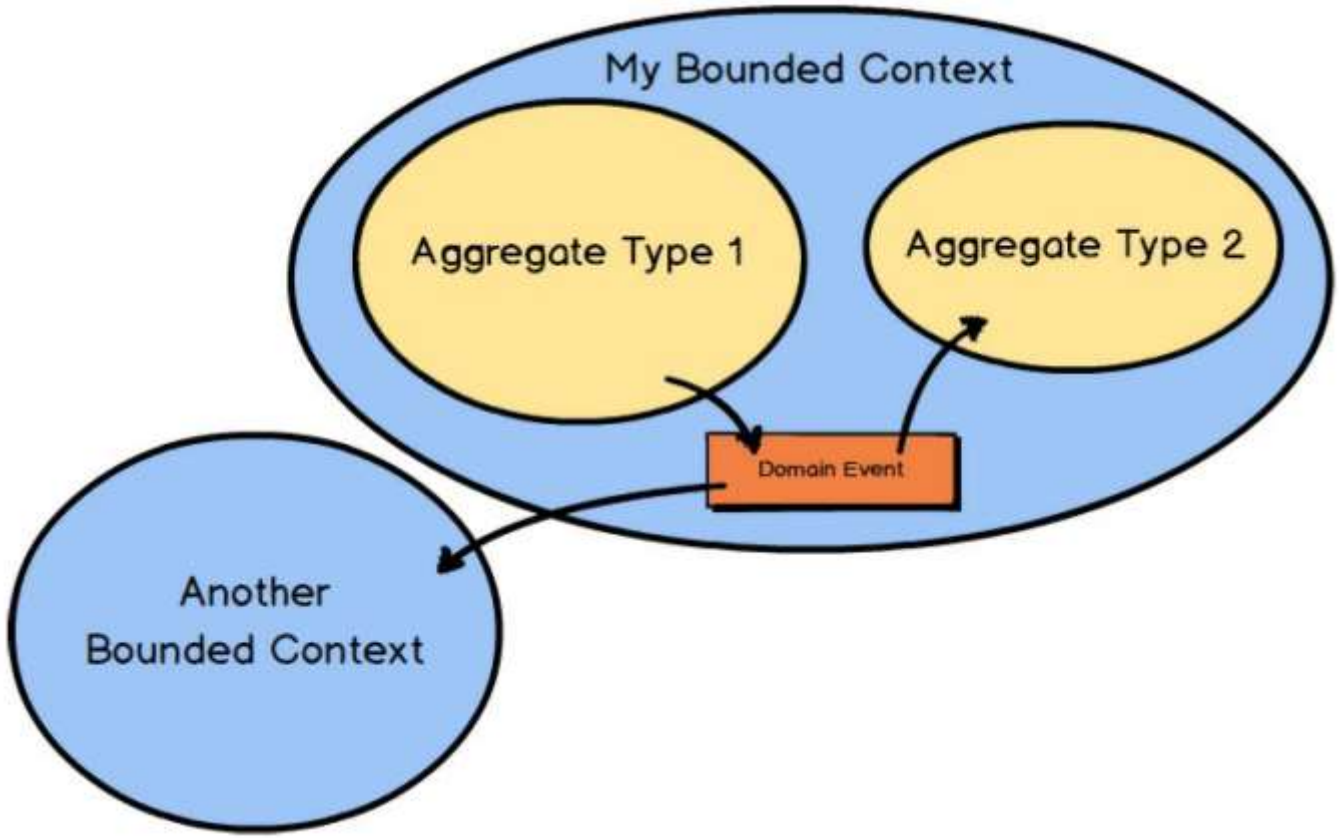
Domain Services

- Hearth of the domain
- Interacts only domain objects(ARs, Entities, ValueObjects) and primitive types(decimal,int, string etc...)
- ApplicationServices must not pass any DTO objects to DomainServices
- Handles only business and domain objects states
- Persistence agnostic(Does not know any DbContext or Nhibernate Session)

Domain Services



Two Bounded Context



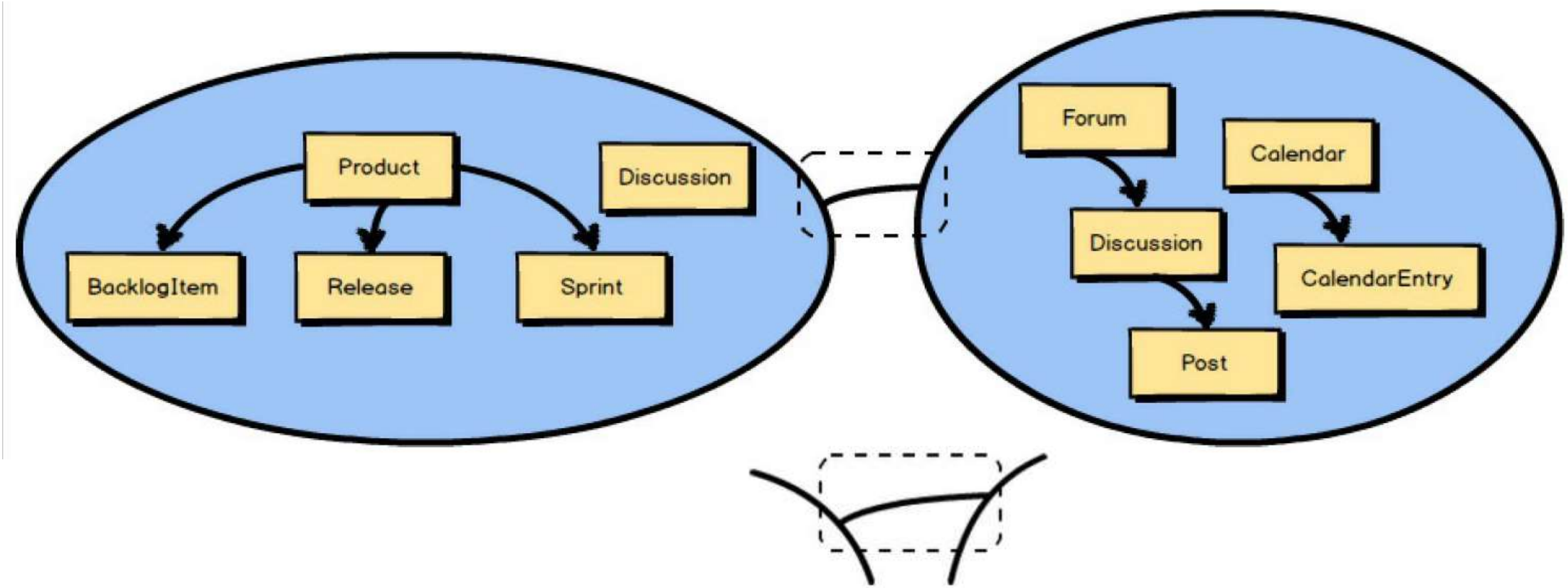
DomainEvents

```
3 references
public class ProductRepository : IRepository<Product>
{
    private readonly IEventPublisher _bus;
    private readonly ConcurrentBag<Product> _db = new ConcurrentBag<Product>();

    2 references
    public ProductRepository(IEventPublisher bus)
    {
        _bus = bus;
    }

    3 references
    public Task<Product> Get(Func<Product, bool> expression)
    {
        Product product = _db.FirstOrDefault(expression);

        return Task.FromResult(product);
    }

    2 references
    public Task<List<Product>> GetAll()
    {
        return Task.FromResult(_db.ToList());
    }

    1 reference
    public Task<List<Product>> GetAll(Func<Product, bool> expression)
    {
        return Task.FromResult(_db.Where(expression).ToList());
    }

    2 references
    public Task Save(Product aggregate)
    {
        ImmutableList<Event> events = aggregate.GetUncommittedEvents();

        _db.Add(aggregate);

        events.ForEach(@event => _bus.Publish(@event));

        aggregate.MarkEventsAsHandled();

        return Task.FromResult(0);
    }
}
```

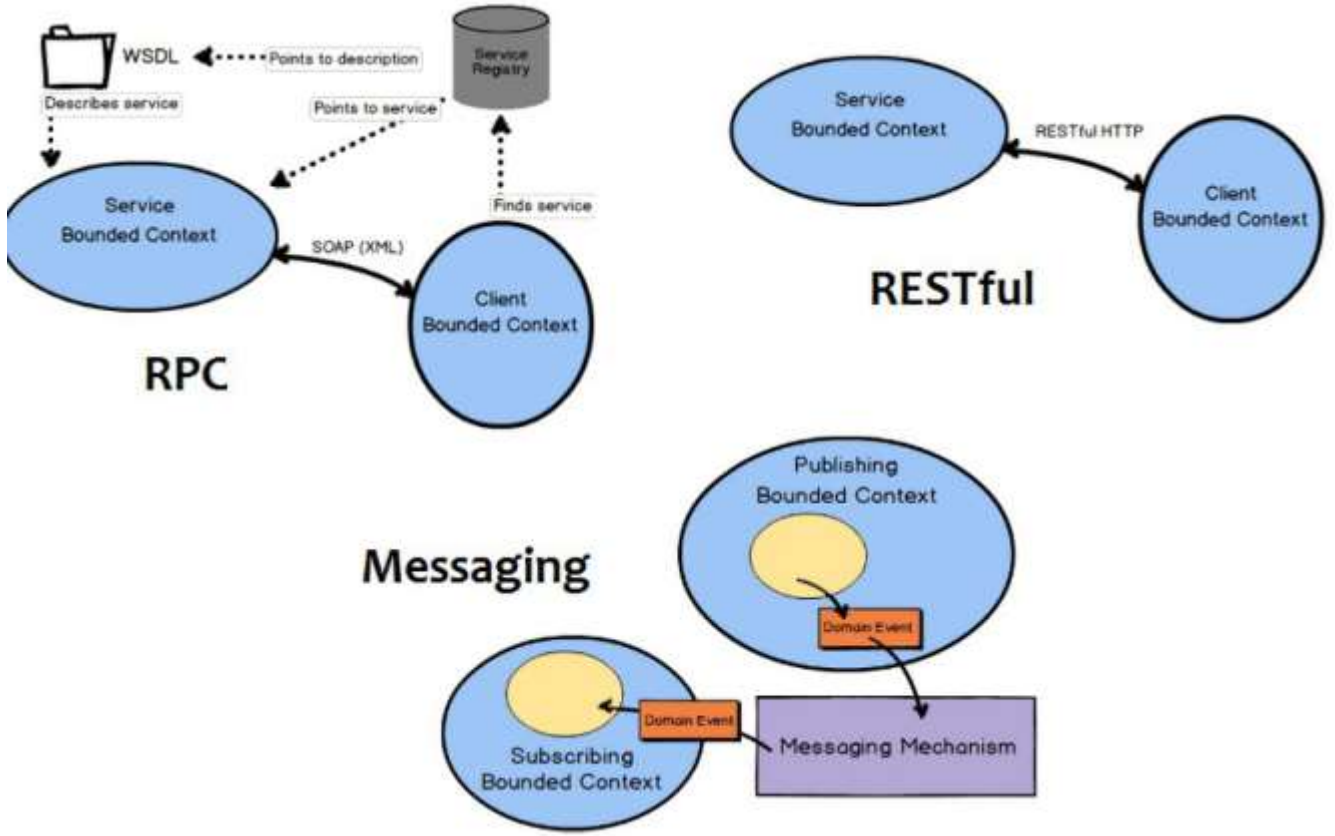
```
5 references
public class ProductCreatedEvent : Event
{
    public string Barcode;
    public string Code;
    public string Name;

    1 reference
    public ProductCreatedEvent(string barcode, string code, string name)
    {
        Barcode = barcode;
        Code = code;
        Name = name;
    }
}
```

```
0 references
public class ProductCreatedEventHandler : IHandles<ProductCreatedEvent>
{
    3 references
    public Task Handle(ProductCreatedEvent message)
    {
        //Do some domain event actions ...
        //Pass message to other BoundedContexts or AggregateRoots

        return Task.CompletedTask;
    }
}
```

Communication Between Bounded Contexts



Robust Way of Communication:Messaging



Discussions & Consolidations

Persistence Ignorance

- Not Database/ORM Driven Design, but Domain Driven Design!

Bounded Contextes

- Entities should not be re-used between contextes even they represent same database table.

Aggregate Roots

- Responsible for it's own validity and integrity.
- Can refer other Aggregates by only reference (by their id).
- Should go over aggregate root to access an aggregate member.
- Should be retrieved as a whole object (including collections) from repository (or support lazy load).

Applications

- Create application layer per application (UI).

How to managing exceptions?

- Infrastructure Exceptions (database connection… etc.)
- Application Exceptions (like a null value sent to a method)
- Authorization & Validation Exceptions
- Domain Exceptions (business rules and validations)

Presentation logic vs application logic vs domain logic.

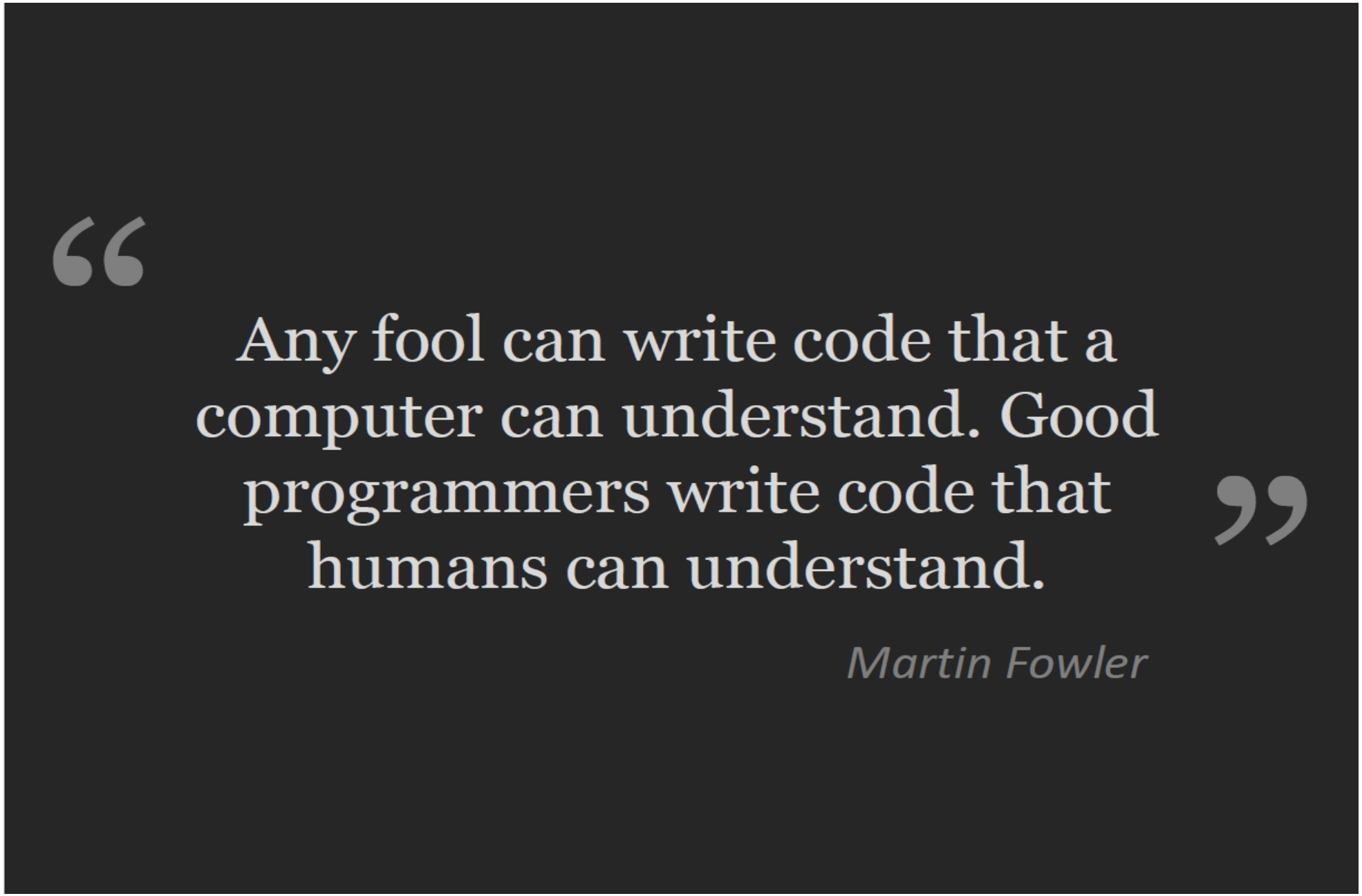
Validation

- UI validation
- DTO validation
- Business validation

Application Service vs Domain Service

Write Model - Read Model / Reporting

- DDD is mostly for changing the state, not reporting!


“
Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
”
Martin Fowler

Thanks!

Source Code: https://github.com/osoykan/Samples/tree/master/DDDCourse

Oğuzhan Soykan