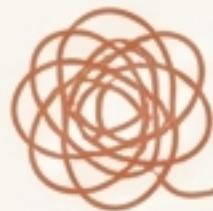


Domain–Driven Design: Teoriden Teoriden Pratiğe, Kavramdan Koda

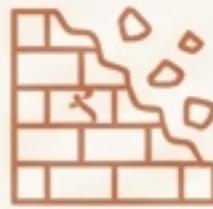
Karmaşık iş mantığını yönetilebilir ve sürdürülebilir
mimarilere dönüştürme sanatı.

Neden DDD'ye İhtiyacımız Var?

Tanıdık Sorunlar



İş kuralları sürekli değişiyor ve kodun her yerine dağılıyor.



Kod zamanla karmaşıklaşıyor, bakımı imkansız hale geliyor.



Geliştiriciler ve iş birimleri aynı dili konuşmuyor, sürekli yanlış anlaşılmalar yaşanıyor.



Nerede hangi iş kuralının uygulandığını bulmak için tüm projeyi gezmek gerekiyor.

Kalıcı Çözüm

“ Sürekli değişen iş yapış kurallarının bile bir ahenk ve düzen içerisinde yazılım projenize başarılı ve derinlemesine uygulanmasını sağlamak...”

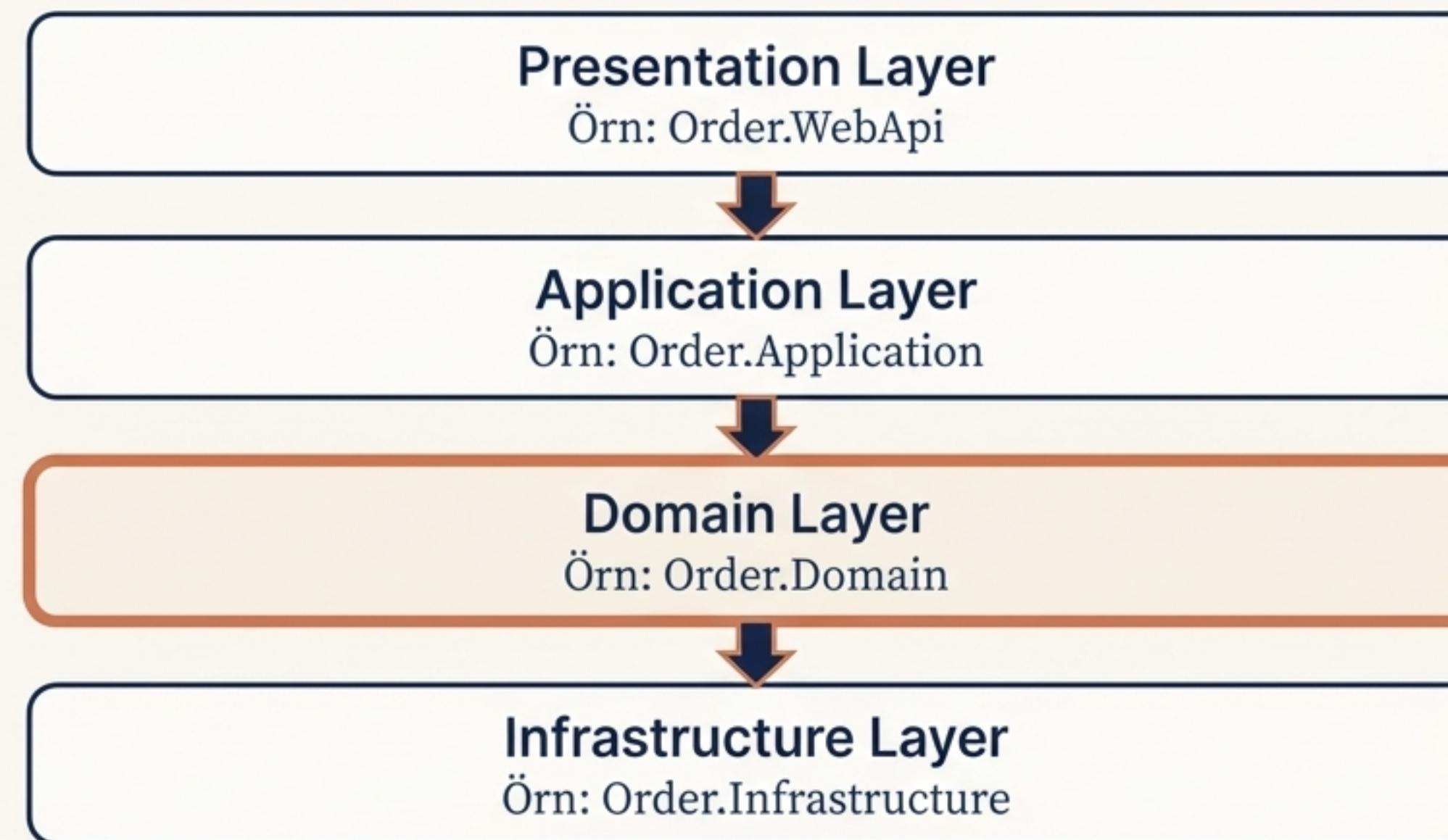
— Eric Evans

DDD, bu karmaşaaya bir düzen ve felsefe getirir.

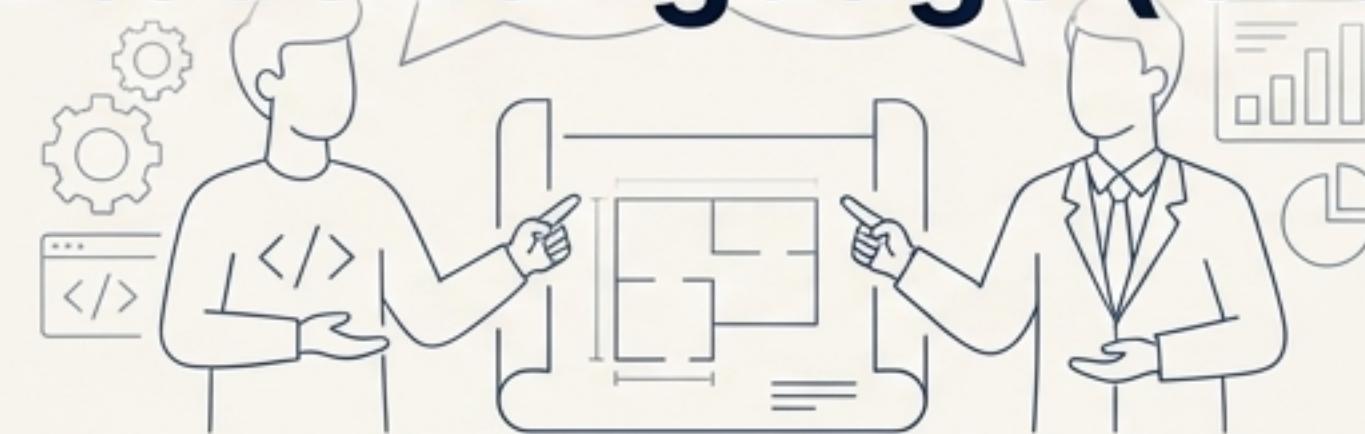
”

Yol Haritamız: Katmanlı Mimari (Layered Architecture)

DDD, temiz ve organize bir yapı üzerine kuruludur. Yolculüğumuza bu temel haritayla başlıyoruz. Anlatacağımız her parça, bu katmanlardan birine hayat verecek.



Her Şey Bir Dille Başlar: Ubiquitous Language (Ortak Dil)



Yazılım ekibi, domain uzmanları ve iş birimlerinin kullandığı ortak, modelin içine **işlenmiş, herkesin üzerinde anlaştığı dildir. Belirsizliğe yer bırakmaz.**

Pratikte Anlamı

- Eğer iş birimi bir kavrama “**Sipariş**” diyorsa, koddaki sınıfın adı `Order` olmalıdır.
- Eğer bir kullanıcı “**Alıcı**” olarak tanımlanıyorsa, ilgili sınıf `Buyer` olarak isimlendirilir.
- Bu dil, sadece sınıflarda değil, metodlarda, değişkenlerde ve veritabanı tablolarında da tutarlı bir şekilde kullanılır.

Anlam Sınırlızılarını Çizmek: Bounded Context

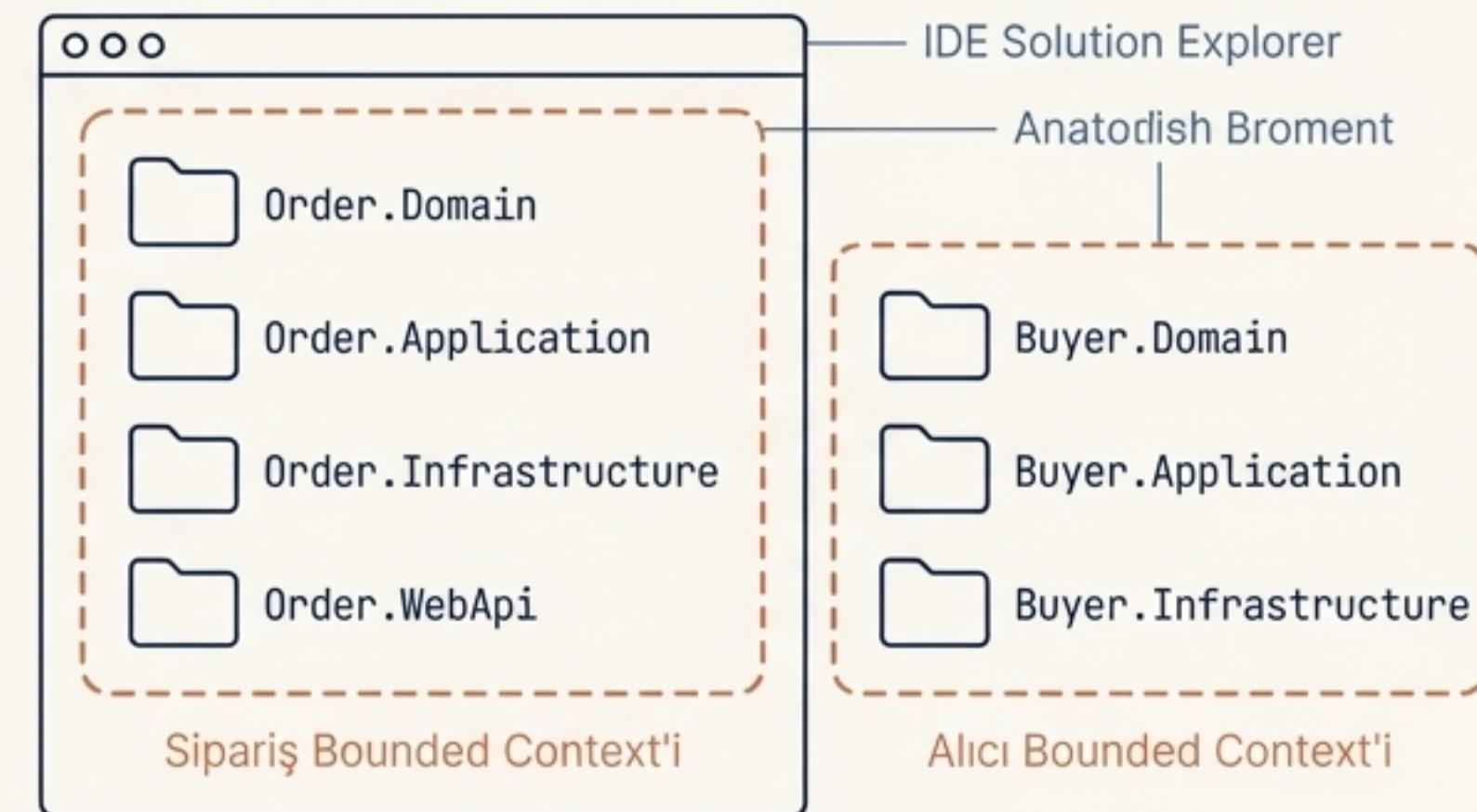
Kavram

Tanım: Bir modelin ve o modelin kullandığı dilin (Ubiquitous Language) tutarlı ve geçerli olduğu sınırlı bir çerçevedir.

- Örnek: Bir e-ticaret sisteminde 'Sipariş', 'Müşteri', 'Stok' ve 'Ödeme' gibi her bir alt domain, kendi Bounded Context'ini oluşturur. 'Sipariş' context'indeki **Product** ile 'Stok' context'indeki **Product** farklı anlamlar ve özellikler taşıyabilir.

Mimarideki Yansımı

Bounded Context, genellikle projenin fiziksel yapısına doğrudan yansır.



Modelin Atomları: Entity ve Value Object

Entity

Bir kimliği (ID) vardır ve yaşam döngüsü boyunca bu kimlikle takip edilir. Değişebilir.
Örnek: Order, Buyer.

Value Object

Kimliği yoktur; onu tanımlayan şey nitelikleridir. Değişmez (immutable).
Örnek: Address.

Entity Code

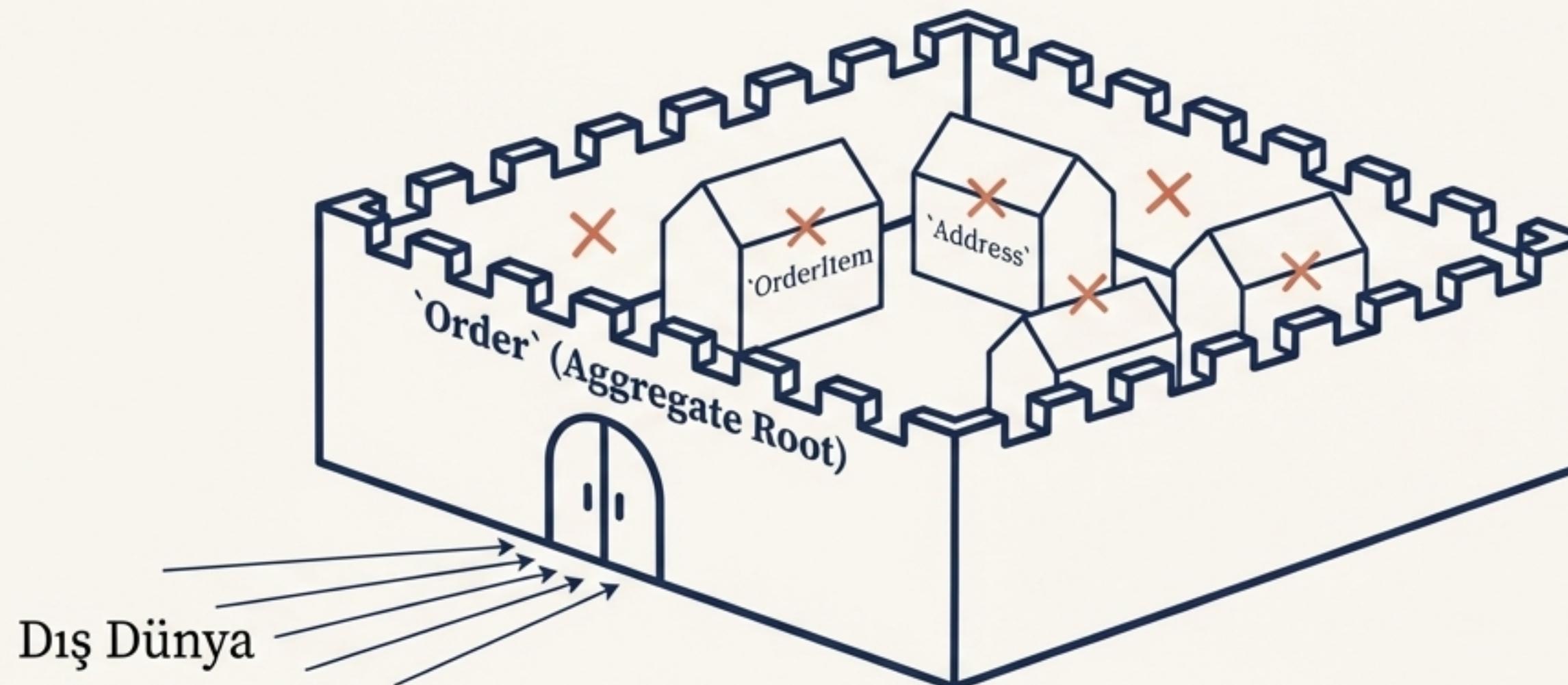
```
// Kimliği olan ve yaşam döngüsü izlenen nesne
public abstract class BaseEntity
{
    public int Id { get; protected set; }
}
public class Order : BaseEntity, IAggregateRoot
{
    // ...
}
```

Value Object Code

```
// Kimliği olmayan, sadece değerleriyle var olan nesne
public class Address : ValueObject
{
    public string Country { get; private set; }
    public string City { get; private set; }
    // ...
}
```

Tutarlılığın Bekçisi: Aggregate Root

Birbiri olmadan anlamı olmayan nesneler grubunu (aggregate) yöneten, dış dünyayla *tek iletişim noktası* olan ve iş kurallarını uygulayan özel bir Entity'dir.



Dış dünya, kalenin içindeki 'OrderItem'lara doğrudan dokunamaz. Her istek, bekçi olan 'Order' üzerinden geçmek zorundadır. Bu, veri tutarlığını garanti eder.

Bekçi Koda Dönüşüyor: `Order` Aggregate'i

```
public class Order : BaseEntity, IAggregateRoot
{
    public DateTime OrderDate { get; private set; }
    public Address Address { get; private set; }

    private readonly List<OrderItem> _orderItems;
    public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;

    public Order(...) { /* ... */ }

    public void AddOrderItem(int productId, decimal price, int quantity)
    {
        // ... kurallar burada işler ...
        var newOrderItem = new OrderItem(productId, price, quantity);
        _orderItems.Add(newOrderItem);
    }
}
```

1. İşaretleyici: Bu sınıfın bir Aggregate Root olduğunu belirten bir işaret.

2. İç Kalenin Sakinleri: `OrderItem` listesi dışarıdan doğrudan değiştirilemez. Kontrol tamamen `Order`da!

3. Tek Giriş Kapısı: Sipariş kalemi eklemek için tek yetkili metot. Tüm iş kuralları ve doğrulamalar bu kapıdan geçerken uygulanır.

İş Kuralları Ait Olduğu Yerde: Kalenin İçinde

Doğrulama (validation) ve iş kuralları, dış servislerde veya arayüz katmanında değil, doğrudan domain modelinin kalbinde, Aggregate Root'un içinde yaşar. Bu, modelin her zaman tutarlı kalmasını sağlar.

```
public Order(DateTime orderDate, Address address, int buyerId)
{
    // Kural 1: Sipariş tarihi geçmiş bir tarih olamaz.
    if (orderDate < DateTime.Now)
        throw new Exception("OrderDate must be greater than today.");
    // Kural 2: Adres bilgisi eksik olamaz.
    if (string.IsNullOrEmpty(address.City))
        throw new Exception("City cannot be empty.");

    this.OrderDate = orderDate;
    this.Address = address;
    // ...
}

public void AddOrderItem(int productId, decimal price, int quantity)
{
    // Kural 3: Miktar sıfırdan büyük olmalıdır.
    if (quantity <= 0)
        throw new Exception("Invalid quantity.");

    // ...
    _orderItems.Add(new OrderItem(...));
}
```

Dünyalar Arası İletişim: Domain Events

Kavram

Bir Bounded Context içinde önemli bir iş olayı gerçekleştiğinde, diğer Bounded Context'leri (veya aynı context'in farklı parçalarını) bundan haberdar etme mekanizmasıdır.

Amaç

Bağlamlar arasında doğrudan ve sıkı bir bağımlılık (tight coupling) kurmaktan kaçınmak. Her bağlam kendi işine odaklanır ve sadece ilgili olayları dinler.

Senaryo



Tepki

Bu olayı duyar ve eğer bu yeni bir alıcıysa, kendi veritabanında yeni bir 'Buyer' kaydı oluşturur.

Sonuç: 'Order' Aggregate'i, 'Buyer' Aggregate'ini veya onun repository'sini doğrudan çağrırmak zorunda kalmaz. İletişim, bir olay aracılığıyla dolaylı olarak sağlanır.

Olaylar Koda Dökülüyor

1. Tanımla → 2. Fırlat → 3. Yakala

- Olayın Tanımlanması ('Domain` Katmanı)

```
// Olayın kendisi: Ne oldu?  
public class OrderStartedEvent : INotification  
{  
    public string UserName { get; }  
    public Order Order { get; }  
  
    public OrderStartedEvent(string userName,  
    Order order) { /* ... */ }  
}
```

- Olayın Fırlatılması ('Domain` Katmanı)

```
// BaseEntity veya Aggregate içinde  
public abstract class BaseEntity  
{  
    private List<INotification> _domainEvents;  
  
    public void AddDomainEvent(INotification  
notification)  
    {  
        _domainEvents.Add(notification);  
    }  
}  
  
// Order'in constructor'ında:  
AddDomainEvent(new OrderStartedEvent(userName,  
this));
```

- Olayın Yakalanması ('Application` Katmanı)

```
// Olayı dinleyip iş yapan Handler  
public class OrderStartedHandler  
    : INotificationHandler<OrderStartedEvent>  
{  
    private readonly IBuyerRepository  
_buyerRepository;  
  
    public async Task Handle(OrderStartedEvent  
n, CancellationToken ct)  
    {  
        // Buyer var mı kontrol et, yoksa yarat.  
        // _buyerRepository.Add(new Buyer(...));  
    }  
}
```

Dış Dünya ile Konuşmak: Repositories ve Infrastructure

- **Kavram:** Repository Pattern, domain modelini veritabanı gibi dış dünyadan ve kalıcılık (persistence) mekanizmalarından soyutlar. Domain katmanı, verilerin nasıl saklandığını bilmez; sadece bir arayüz (IRepository) üzerinden konuşur.
- Source Serif Pro Regular, Deep Navy

Koddaki Rol Dağılımı

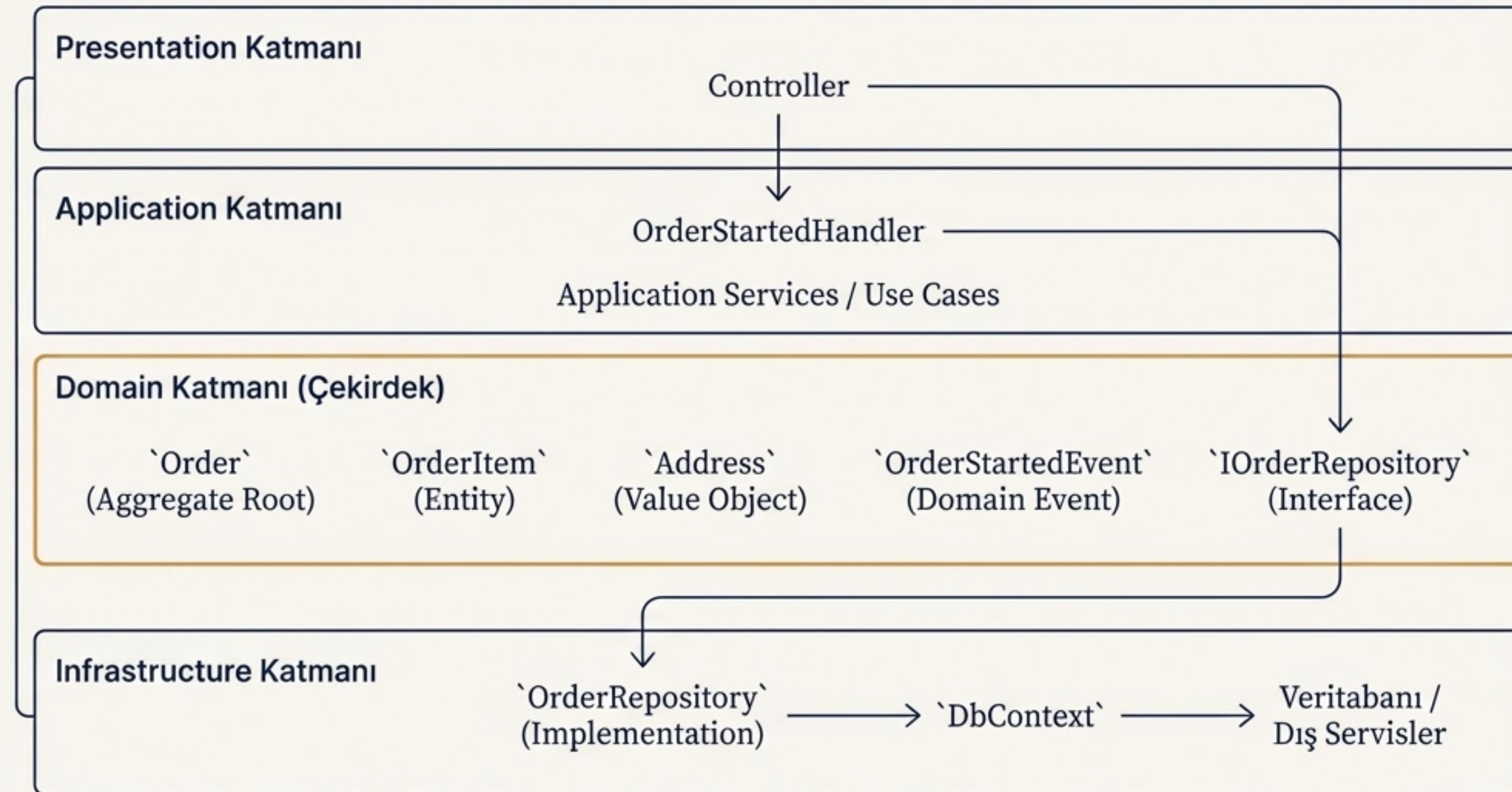
Domain Katmanı (Sözleşme)

```
// Sadece ne yapılacağını söyler, nasıl değil.  
public interface IOrderRepository : IRepository<Order>  
{  
    // Order'a özel metodlar buraya gelebilir.  
    Task<Order> GetByIdAsync(int id);  
}
```

Infrastructure Katmanı (Uygulama)

```
// Veritabanı teknolojisine özel kod burada yaşar.  
public class OrderRepository : IOrderRepository  
{  
    private readonly OrderDbContext _context;  
  
    public async Task<Order> GetByIdAsync(int id)  
    {  
        return await _context.Orders.FindAsync(id);  
    }  
    // ...  
}
```

Büyük Resim: Her Parça Yerine Oturdu



Bir Desenler Bütününden Daha Fazlası: Felsefe ve Zanaatkarlık

Domain-Driven Design, sadece anlık görevleri tamamlamakla ilgili değildir. Kodu sürekli iyileştirmek, daha okunabilir hale getirmek ve objelerin genelini daha sağlıklı kılmak için atılan adımları da içerir.

“Kodunu yazarken şair ol, kendiniz değil başkası okuyacakmış gibi kodlarınızı yazın. Performanslı ama anlaşılmaz kodlar yazmak yerine daha basit ve anlaşılabilir kodlar yazmak genelde herkes için faydalı olacaktır.”

Özet ve Sonraki Adımlar

- **Ortak Dil (Ubiquitous Language) ile Konuş:** İş birimleri ve geliştiriciler arasında köprü kur. İsimlendirmelerin bilinçli olsun.
- **Sınırlarını (Bounded Context) Çiz:** Modellerini anlamlı bağamlara ayırarak karmaşıklığı yönet. Bu sınırları mimarine yansıt.
- **Bekçilerini (Aggregate Root) Oluştur:** Veri tutarlığını garanti altına al ve iş kurallarını ait oldukları yere, domain modelinin kalbine koy.
- **Olaylar (Domain Events) ile Haberleş:** Bağamların arasında esnek ve zayıf bağımlı bir iletişim mimarisi kur.

Bu prensipleri bir sonraki adımda mikroservis mimarisinde nasıl uyguladığımızı görmek için takipte kalın!