**CODE
PROJECT**
For those who code

Enter for a chance to win \$500 cash!
Take a 2 min survey to have your voice heard, earn points that lead to cash prizes, and be entered in a \$500 drawing!

Take Survey

[articles](#) [quick answers](#) [discussions](#) [features](#) [community](#) [help](#)

Search for articles, questions, tips



Articles » Development Lifecycle » Design and Architecture » Patterns and Practices



Domain-Driven Design, My Top 5 Best Practices

**Z Kavtaskin**24 Sep 2016 [CPOL](#)Rate this:  5.00 (7 votes)

Domain driven design - my top 5 best practices

Introduction

About 3 years ago, I wrote [Applied Domain-Driven Design Series](#) and created a ["Domain-Driven Design Example" Github repository](#).

I've noticed that we all make the same mistakes as we learn DDD. In this article, I am going to try and address some of the most common mistakes that I have seen or made. This best practice list is limited, please also check out ["10 Domain-Driven Design Mistakes"](#) by Daniel Whittaker.

1. Always Start with the Requirements & Modeling

I remember when I first discovered the Eric Evan's blue book. I read it and loved it. However, I was mostly excited about leaving behind the anemic models, introducing Aggregate Roots, Domain Services, etc. I missed the main point of the Domain-driven design. The point is that you should be immersing yourself in the business requirements, asking lots of questions about the business and modeling together with the business analysts on the whiteboard. Instead, I have ended up modeling my own non business focused technical reality. This model was not maintainable long term and had to be re-written. [Avoid this by always starting with the requirements and modeling.](#)

2. Keep Your Aggregate Roots Flat Whenever Possible

When I first started using DDD, I was very excited by the idea of deep object graph nesting. Imagine that you have an object **Customer**, **customer** object can access **Cart**, **Purchases**, **Credit Cards**, etc. **Purchases** can access **items** that were purchased and so on. It was exciting because I could use **Specification pattern** and traverse the hell out of my model. This was exciting because it enabled me to write complex domain queries and I could project the query output onto a very complex UI. Unfortunately, very quickly, it made unit testing very difficult and it made the application hard to maintain.

Here is an example of deep object graph:

[Hide](#) [Shrink ▲](#) [Copy Code](#)

```
public class Customer : IAggregateRoot
{
    private List<Purchase> purchases = new List<Purchase>();
    private List<CreditCard> creditCards = new List<CreditCard>();

    public virtual Guid Id { get; protected set; }
    public virtual string FirstName { get; protected set; }
    public virtual string LastName { get; protected set; }
    public virtual string Email { get; protected set; }
    public virtual string Password { get; protected set; }
    public virtual DateTime Created { get; protected set; }
    public virtual bool Active { get; protected set; }
    public virtual decimal Balance { get; protected set; }
    public virtual Country Country { get; protected set; }

    public virtual ReadOnlyCollection<Purchase> Purchases
    { get { return this.purchases.AsReadOnly(); } }
    public virtual ReadOnlyCollection<CreditCard> CreditCards
    { get { return this.creditCards.AsReadOnly(); } }

    public virtual Cart Cart { get; protected set; }

    public virtual void ChangeEmail(string email)
    {
        if(this.Email != email)
        {
            this.Email = email;
            DomainEvents.Raise<CustomerChangedEmail>(new CustomerChangedEmail() { Customer = this });
        }
    }

    public static Customer Create(string firstname, string lastname, string email, Country country)
    {
        return Create(Guid.NewGuid(), firstname, lastname, email, country); ;
    }
}
```

```
public static Customer Create(Guid id, string firstname, string lastname, string email, Country country)
{
    if (string.IsNullOrEmpty(firstname))
        throw new ArgumentNullException("firstname");

    if (string.IsNullOrEmpty(lastname))
        throw new ArgumentNullException("lastname");

    if (string.IsNullOrEmpty(email))
        throw new ArgumentNullException("email");

    if (country == null)
        throw new ArgumentNullException("country");

    Customer customer = new Customer()
    {
        Id = id,
        FirstName = firstname,
        LastName = lastname,
        Email = email,
        Active = true,
        Created = DateTime.Today,
        Country = country
    };

    DomainEvents.Raise<CustomerCreated>(new CustomerCreated() { Customer = customer });

    customer.Cart = Cart.Create(customer);

    return customer;
}

public virtual ReadOnlyCollection<CreditCard> GetCreditCardsAvailble()
{
    return this.creditCards.FindAll
        (new CreditCardAvailableSpec(DateTime.Today).IsSatisfiedBy).AsReadOnly();
}

public Nullable<PaymentIssues> IsPayReady()
{
    if (this.Balance < 0)
        return PaymentIssues.UnpaidBalance;

    if (this.GetCreditCardsAvailble().Count == 0)
        return PaymentIssues.NoActiveCreditCardAvailable;

    return null;
}
```

```

public virtual void Add(CreditCard creditCard)
{
    this.creditCards.Add(creditCard);

    DomainEvents.Raise<CreditCardAdded>(new CreditCardAdded() { CreditCard = creditCard });
}

internal virtual void Add(Purchase purchase)
{
    this.purchases.Add(purchase);
}
}

```

If I were to refactor the above, what would I change?

- Remove **Purchases** and **County** properties. I will never need to access **County** properties, so why reference it? List of purchases will not be accessed every time I load a **Customer**, so there is really no point having it on the **customer**. Yes, you could go with the lazy loading. But why have properties that you don't really need on the object in the first place?
- Move and refactor **IsPayReady**. **IsPayReady** is part of the checkout context. I don't think it should sit inside the **customer**. So I have created **CheckoutService**, this is a domain service that checks if **customer** can pay and **product** can be sold, it then performs the **purchase**.
- Remove **Cart** property. **Cart** is an aggregate root and can be accessed directly when it's needed, once again, there is no need to load it every time with the **Customer**.
- Remove **Created** and **Active** properties. Unless your business analysts say that you need to have these two properties in your domain as they are part of the business domain, you should not have them. Also, I've introduced **Domain Event Logging** to my codebase so there is really no need to pollute your Domain Model with the object lifecycle information as this is already captured in the events.

Here is how **Customer** aggregate root looks like now:

Hide Shrink ▲ Copy Code

```

public class Customer : IAggregateRoot
{
    private List<CreditCard> creditCards = new List<CreditCard>();

    public virtual Guid Id { get; protected set; }
    public virtual string FirstName { get; protected set; }
    public virtual string LastName { get; protected set; }
    public virtual string Email { get; protected set; }
    public virtual string Password { get; protected set; }
    public virtual decimal Balance { get; protected set; }
    public virtual Guid CountryId { get; protected set; }

    public virtual ReadOnlyCollection<CreditCard> CreditCards
    { get { return this.creditCards.AsReadOnly(); } }

    public virtual void ChangeEmail(string email)
    {
        if(this.Email != email)
        {

```

```
        this.Email = email;
        DomainEvents.Raise<CustomerChangedEmail>(new CustomerChangedEmail() { Customer = this });
    }
}

public static Customer Create(string firstname, string lastname, string email, Country country)
{
    return Create(Guid.NewGuid(), firstname, lastname, email, country); ;
}

public static Customer Create
(Guid id, string firstname, string lastname, string email, Country country)
{
    if (string.IsNullOrEmpty(firstname))
        throw new ArgumentNullException("firstname");

    if (string.IsNullOrEmpty(lastname))
        throw new ArgumentNullException("lastname");

    if (string.IsNullOrEmpty(email))
        throw new ArgumentNullException("email");

    if (country == null)
        throw new ArgumentNullException("country");

    Customer customer = new Customer()
    {
        Id = id,
        FirstName = firstname,
        LastName = lastname,
        Email = email,
        CountryId = country.Id
    };

    DomainEvents.Raise<CustomerCreated>(new CustomerCreated() { Customer = customer });

    return customer;
}

public virtual ReadOnlyCollection<CreditCard> GetCreditCardsAvailble()
{
    return this.creditCards.FindAll
        (new CreditCardAvailableSpec(DateTime.Today).IsSatisfiedBy).AsReadOnly();
}

public virtual void Add(CreditCard creditCard)
{
    this.creditCards.Add(creditCard);
}
```

```
DomainEvents.Raise<CreditCardAdded>(new CreditCardAdded() { CreditCard = creditCard });  
}  
}
```

Why is it a good idea to keep aggregate roots flat?

- Better performance as you load less data
- Easier to unit test
- Easier to maintain and extend
- Requires less ORM configuration
- Smaller transactions, mean smaller locks, which means better throughput

For this article, I have refactored my "[Domain-Driven Design Example](#)" Repository on Github. If you would like to see all of the changes, [click here](#).

3. Domain Model != Read Model

When it comes to Domain model, it can be awkward to query. I remember when a requirement came through that I needed to project some more data onto the UI. It was so painful. I had to extend my aggregate roots and add extra properties onto it, which violated best practice number 1 and 2. To make it performant, I've spent hours tuning NHibernate config. This was a wrong thing to do. I should have used my aggregate roots just for commands and I should have used database views to project data out. Reason I have done that was because I was thinking that with DDD I should use Domain model for everything. Just use Specification pattern and traverse your model, problem solved, SQL is dead and Specification pattern is the way forward. This was naive.

For complex API or UI projections, you should be using **Read** model. **Read** model is not a scary thing, it can be something as simple as a database view. If you would like to see more examples, [please click here](#).

Also, I recommend that when developers start using DDD, they avoid Specification Pattern, it does more damage than good.

4. Domain Events are Immutable Facts

Many people don't fully understand domain events. They think that domain event is an interactive interface. **Domain** event **CustomerCheckedOut** is fired. Handler picks it up, loads another aggregate root and that aggregate root can reject domain event **CustomerCheckedOut**. **Domain** event is not an interactive interface. **Domain** event is just an immutable fact. **Domain** event handlers listen for the event, all these handlers can do is just change other aggregate roots, send out emails, etc. Handler can't change the domain event and they can't change the aggregate root that has fired the domain event. You should not use **domain** events to ask for permission to do something, e.g., **CanCustomerCheckout** domain event asks the whole systems, can this **customer** checkout, is it OK?

When would you use **Domain** Events? When you need to tell the rest of your **Domain** model that something has already happened, it's an opportunity for some handler to act upon it. For example, a handler would send out an email, it would load some business counter and increment it for analytics, it would re-sync some bounded context data.

5. Aggregate Roots Can't Access Repositories

As a rule of thumb, only Application Services, Domain Services and Domain Event Handlers can access your repository interfaces. Aggregate roots should not access repositories.

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOl\)](#)



Share



About the Author



Z Kavtaskin

 United Kingdom 

No Biography provided

Comments and Discussions

You must [Sign In](#) to use this message board.



Spacing

Relaxed ▼

Layout

Normal ▼

Per page

25 ▼

Update

-- There are no messages in this forum --

[Permalink](#)[Advertise](#)[Privacy](#)[Cookies](#)[Terms of Use](#)Layout: [fixed](#) | [fluid](#)

Article Copyright 2016 by Z Kavtashin
Everything else Copyright © [CodeProject](#), 1999-2020

Web03 2.8.200331.1