ZAN KAVTASKIN

Musings about Software Engineering

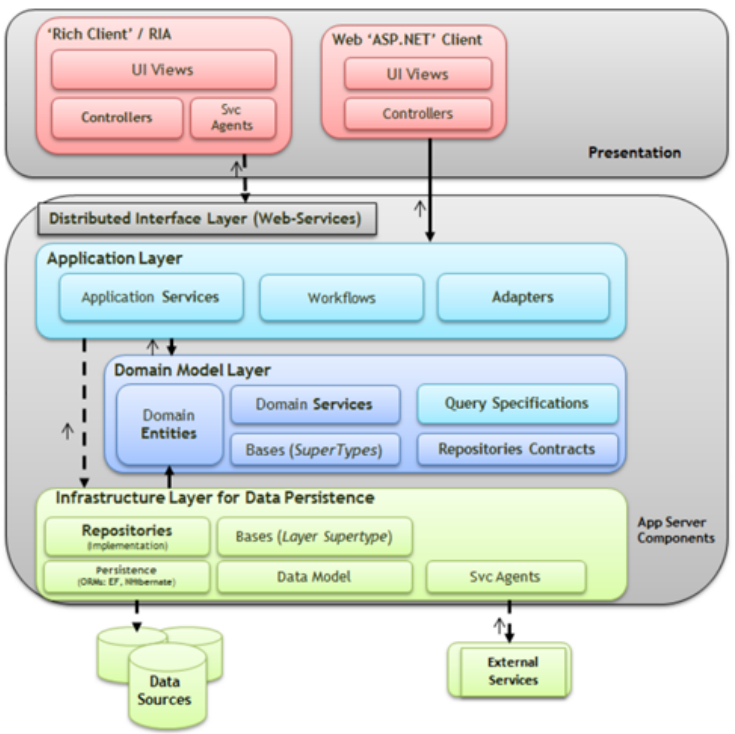
Thursday, 26 September 2013

Applied Domain-Driven Design (DDD), Part 1 - Basics

When I started learning domain-driven design there was a lot of theory to take in, Eric Evans did a great job explaining it from theoretical point of view. As a software engineer I wanted to see some code and just to follow some examples, I found very little resource out there that showed applied domain-driven design in C#.

Over the coming weeks I will be posting series of articles on the subject, it will be my attempt to make domain-driven design simpler and easier follow. Articles that are published:

- Applied Domain-Driven Design (DDD), Part 0 - Requirements and Modelling
- Applied Domain-Driven Design (DDD), Part 1 - Basics
- Applied Domain-Driven Design (DDD), Part 2 - Domain events
- Applied Domain-Driven Design (DDD), Part 3 - Specification Pattern
- Applied Domain-Driven Design (DDD), Part 4 - Infrastructure
- Applied Domain-Driven Design (DDD), Part 5 - Domain Service
- Applied Domain-Driven Design (DDD), Part 6 - Application Services
- Applied Domain-Driven Design (DDD), Part 7 - Read Model
- Applied Domain-Driven Design (DDD), My Top 5 Best Practices
- Applied Domain-Driven Design (DDD), Event Logging & Sourcing For Auditing



Domain Driven Design Architecture (it's simpler then it looks)

Blog Archive

- ▶ 2020 (2)
- ▶ 2019 (1)
- ▶ 2018 (7)
- ▶ 2017 (5)
- ▶ 2016 (9)
- ▶ 2014 (3)
- ▼ 2013 (9)
  - ▶ Dec (3)
  - ▶ Nov (3)
  - ▶ Oct (1)
  - ▼ Sep (2)
    - Applied Domain-Driven Design (DDD), Part 2 - Domai...
    - Applied Domain-Driven Design (DDD), Part 1 - Basic...

Before we get started let's see why DDD is so great:

- **Development** becomes domain oriented not UI/Database oriented
- **Domain layer** captures all of the business logic, making your service layer very thin i.e. just a gateway in to your domain via DTO's
- **Domain oriented development** allows you to implement true service-oriented architecture i.e. making your services reusable as they are not UI/Presentation layer specific
- **Unit tests** are easy to write as code scales horizontally and not vertically, making your methods thin and easily testable
- **DDD** is a set of Patterns and Principles, this gives developers a framework to work with, allowing everyone in the development team to head in the same direction

Through this series of articles I will be focusing on a simple and made up e-commerce domain.

So, let's see some code:

```
public class Product
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public int Quantity { get; set; }
```

```
        public DateTime Created { get; set; }
        public DateTime Modified { get; set; }
        public bool Active { get; set; }
    }

    public class Customer
    {
        public Guid Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Email { get; set; }
        public List Purchases { get; set; }
}

    public class Purchase
    {
        public Guid Id { get; set; }
        public List Products { get; set; }
        public DateTime Created { get; set; }
        public Customer Customer { get; set; }
    }
```

Code above represents anemic classes. Some developers would stop here, and use these classes to pass data to service and then bind this data to the UI. Let's carry on and mature our models.

When a customer shops online they choose items first, they browse around, and then eventually they will make a purchase. So we need something that will hold the products, lets call it a Cart, this object will have no identity and it will be transient.

Cart is my value object:

```
    public class Cart
    {
        public List Products { get; set; }
    }
```

Cart simply contains a list of products. Customer can go ahead and checkout these products when they are ready.

We can use what was said above as a business case "Customer can go ahead and checkout these products when they are ready".

Code would look like:

```
            Cart cart = new Cart()
            {
                Products = new Product[]
                {
                    new Product(),
                    new Product()
```

```
            }
        };

        Customer customer = new Customer()
        {
            FirstName = "Josh",
            LastName = "Smith"
        };

        Purchase purchase = customer.Checkout(cart);
```

What's going on here? Customer checks-out the product and gets a purchase back. Normally in business context you get a receipt back, this provides basic information about the purchase, discounts, and acts as a guarantee that i can use to refer back to my purchase.

I could rename Purchase to Receipt, but wait, what's does purchase mean in the business context?

> *"to acquire by the payment of money or its equivalent; buy."* - Dictionary.com

(Returning purchase object would make sense if customer actually made a purchase i.e. we pre-authenticated a customer and then simply passed payment authentication code to the checkout, but to keep this simple we are not going to do this)

Customer can now checkout, when they checkout they will get back a Purchase object (this will allow further data manipulation).

Code above needs to be re-factored, if we return back a purchase are we going to then add it to the collection of the customer i.e. Customer.Purchases.Add(...)? This seems strange, if we have a method Customer.Checkout(...) we should aim to capture relevant data right there and then. Customer should only expose business methods, if we have to expose something else in order to capture data then we are not doing it right.

Lets refine further:

```
public class Customer
{
    public Guid Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public List Purchases { get; set; }

    public Purchase Checkout(Cart cart)
    {
        Purchase purchase = new Purchase()
        {
             Customer = this,
             Products = cart.Products,
             Created = DateTime.Now
        };
```

```
            this.Purchases.Add(purchase);
            return purchase;
        }
    }
```

Ok, so now when customer checks-out a cart, purchase will be added to the purchase collection and also returned so it can be used further in our logic. This is great, but another software engineer can go in and compromise our domain. They can just add Orders directly to the customer without checking out i.e. Customer.Orders.Add(...).

Lets refine further:

```
    public class Customer
    {
        private List purchases = new List();

        public Guid Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Email { get; set; }
        public ReadOnlyCollection Purchases { get { return this.purchases.AsReadOnly(); } }

        public Purchase Checkout(Cart cart)
        {
            Purchase purchase = new Purchase()
            {
                 Customer = this,
                 Products = cart.Products,
                 Created = DateTime.Now
            };
            this.purchases.Add(purchase);
            return purchase;
        }
    }
```

Now orders can't be compromised, and code forces software engineers to checkout a cart. What about other properties? They are not protected. We know customer state can't just be changed, we have to go through a process. You need to ask your self, when personal information is changed on the customer, do we need to send an email out? Do we need to call some 3rd party API to sync up our records? Right now you might not have a requirement from your business analysts to do anything like this, so lets not worry about it, lets just protect these fields so they can't be changed.

```
    public class Customer
    {
        private List purchases = new List();

        public Guid Id { get; protected set; }
        public string FirstName { get; protected set; }
```

```
    public string LastName { get; protected set; }
    public string Email { get; protected set; }
    public ReadOnlyCollection Purchases { get { return this.purchases.AsReadOnly(); } }
    public Purchase Checkout(Cart cart)
    {
        Purchase purchase = new Purchase()
        {
             Customer = this,
             Products = cart.Products,
             Created = DateTime.Now
        };
        this.purchases.Add(purchase);
        return purchase;
    }
}
```

That's great, now other software engineers in the team can't change personal information without adding a new method such as Customer.ChangeDetails(...).

Taking in to account what was said above, thinking process, constant re-factoring and making the model match the actual business domain, this is what I've got so far:

```
public class Product
{
    public Guid Id { get; protected set; }
    public string Name { get; protected set; }
    public int Quantity { get; protected set; }
    public DateTime Created { get; protected set; }
    public DateTime Modified { get; protected set; }
    public bool Active { get; protected set; }
}

public class Cart
{
    private List products;

    public ReadOnlyCollection Products
    {
        get { return products.AsReadOnly(); }
    }

    public static Cart Create(List products)
    {
        Cart cart = new Cart();
        cart.products = products;
        return cart;
```

```
    }
}

public class Purchase
{
    private List products = new List();

    public Guid Id { get; protected set; }
    public ReadOnlyCollection Products
    {
        get { return products.AsReadOnly(); }
    }
    public DateTime Created { get; protected set; }
    public Customer Customer { get; protected set; }

    public static Purchase Create(Customer customer, ReadOnlyCollection products)
    {
        Purchase purchase = new Purchase()
        {
            Id = Guid.NewGuid(),
            Created = DateTime.Now,
            products = products.ToList(),
            Customer = customer
        };
        return purchase;
    }
}

public class Customer
{
    private List purchases = new List()

    public Guid Id { get; protected set; }
    public string FirstName { get; protected set; }
    public string LastName { get; protected set; }
    public string Email { get; protected set; }
    public ReadOnlyCollection Purchases { get { return this.purchases.AsReadOnly(); } }

    public Purchase Checkout(Cart cart)
    {
        Purchase purchase = Purchase.Create(this, cart.Products);
        this.purchases.Add(purchase);
        return purchase;
    }

    public static Customer Create(string firstName, string lastName, string email)
    {
        Customer customer = new Customer()
        {
```

```
                FirstName = firstName,
                LastName = lastName,
                Email = email
            };
            return customer;
        }
    }
```

Example usage:

```
            Cart cart = Cart.Create(new List() { new Product(), new Product() });
            Customer customer = Customer.Create("josh", "smith", "josh.smith@microsoft.com");
            Purchase purchase = customer.Checkout(cart);
```

Found this useful?
Browse "Domain-Driven Design Example" Repository On Github.

Summary:

- DDD is all about capturing business logic in the domain i.e. entities, aggregate roots, value objects and domain service.
- DDD is all about thought process and challenging where should what go and what is most logical.
- DDD is all about constant re-factoring and maturing your model as you get further requirements. More requirements your receive the better and stronger your domain will be. Therefore requirements are gold and something that software engineers should always strive to understand.

Useful links:

- Onion architecture, a very good example of what domain-driven design is all about
- Aggregate root, great explanation of what aggregate root actually is
- Explanation of Aggregate Root, Entity and Value objects
- Supple Design, Intention revealing interfaces, etc. Pdf from University of Colorado

**Note: Code in this article is not production ready and is used for prototyping purposes only. If you have suggestions or feedback please do comment.*

-
-
-
-
-

Rate this blog post (72 Votes)

Posted by Zan Kavtaskin
Labels: domain-driven design, software engineering

20 comments:

Unknown 23 May 2014 at 02:04

great posts, thanks

Reply

Replies

Zan Kavtaskin 26 May 2014 at 06:13

Thanks for reading!

Reply

bmoc 14 August 2014 at 23:54

Great post!
We are trying to use DDD approach, but I have some uncertainty in the field of architecture, I would be graceful I you could resolve my doubts :) .
Could you explain more the DDD Architecture diagram in context of dependencies (what layers depends on what layers), please ?
When reading from the to to the bottom, I assume that the higher Layer is dependent on the lower layer, but not the opposite. The exception is with infrastructure layer - IT Uses Domain Layer, but the domain Doesn't know about infrastructure
I can see that

1. The Presentation layer, USES (is dependent on): Distributed Interface Layer (when we have one) or Application Layer.
2. The Application Layer uses
a) Domain Layer (but Domain Layer doesn't know nothing about the application layer).
b) Infrastructure layer (infrastructure Layer doesn't know about App layer)
3. The infrastructure layer Uses
- Domain Layer
- Access Data Services and External Services
Do I understand it correctly ?
Where would you put DataTransferObjects (DTOs) ? Application Layer ? Infrastructure Layer ? Both ?
DTO in Infrastructure:
I can imagine situation when I want DTO to be available in Infrastructure Layer, because I want to pass some complicated view (perspective that is not domain object) directly to the Presentation layer (when App Layer Asks directly repository implementation) ? Is it violation of this architecture ?
DTO in Application:
I don't want to expose my Domain Model directly to the Presentation (for a number of reasons), so create DTOs in Application Layer, and transform Domain Objects to Dto in Application layer.

Reply

Replies

Zan Kavtaskin 1 September 2014 at 05:22

Thanks for the comment!

1. & 2. Yes
3. This is a great question. I think there are several questions here.

Let's start with the infrastructure layer.

You "Domain Model Layer" should not be even aware of the "Infrastructure Layer". You achieve this by putting interfaces (or you can call them contracts, as per diagram) for data persistence (repositories) and any other infrastructure interfaces in to the "Domain Model Layer".
The actual implementation of the repositories sits in the infrastructure layer.

Your infrastructure layer will need to reference "Domain Model Layer" so that Infrastructure knows what entities / objects it's dealing with.

You don't need to have DTOs in the infrastructure layer. It's unnecessary complexity. For a view or something that is not part of a domain you could create a "value object". Please see
http://domainlanguage.com/ddd/patterns/DDD-Pattern-Language-Overview-sml.png (express state & computation with value objects).

Data Transfer Objects i.e. DTOs are fantastic because they don't contain any logic or methods they just carry data. Also if you use an ORM like nHibernate you will notice that it uses reflection to add extra behavior to the object. Last thing you want to do is send an entity with state tracking and methods with logic to the presentation layer. This is why they are so awesome (you have mentioned this above as well).

So, if you want clean separation I would query the data persistence, get back the entity or value object and map it on to a DTO via AutoMapper. Personally I would always do this even for a simple view query, this will keep everything consistent.

For some code samples please see:
https://github.com/zkavtaskin/Domain-Driven-Design-Example

I hope this helps!

Reply

Alex Aldazabal 8 November 2015 at 08:50

Great post!!, It helped me a lot to understand more about DDD!

Reply

Replies

Zan Kavtaskin 23 November 2015 at 13:59

Thanks!

Reply

kso 2 August 2016 at 17:17

Great post.
Could you explain why Development becomes domain oriented not UI/Database oriented ? why not Database orianted?

Reply

Replies

Zan Kavtaskin 4 August 2016 at 15:01

Thank you for the comment!

Domain-driven design makes your development more domain oriented and technology agnostic due to abstraction. Take a look at the onion architecture http://jeffreypalermo.com/blog/the-onion-architecture-part-1/.

Some “classic” applications are very database oriented. For example some of these apps are all about stored procedures. To make things simpler, people just project stored procedure data output directly onto the UI. Overtime this creates maintenance nightmare as these stored procedures often get used by other stored procedures and they start to contain more and more business logic as business processes mature. Your user interface gets intertwined with database, database gets intertwined with business logic. Things become extremely hard to change and understand. No one understands where business process begins and ends.

N-tier architecture removes this confusing business logic and UI intertwining. Domain-driven design dramatically improves maintainability through very good use of object-oriented programming and abstraction from infrastructure concerns.

I hope this answer your question, once again thank you for reading.

Reply

Unknown 20 December 2016 at 18:36

The N-Tier App Design in C# course on Pluralsight demonstrates how to move from a data-centric to a domain-centric design: https://www.pluralsight.com/courses/n-tier-apps-part1

Reply

Unknown 4 May 2017 at 02:27

I want to say thanks for the whole articles you wrote,it's hard to find some explanation or elaboration for such topic of concept in Taiwan. I spent a lot of time learning DDD and onion architecture,and through these articles give me better understanding of that,just wanna to say thanks.

Reply

Replies

Zan Kavtaskin 27 July 2017 at 15:47

Thank you very much, I really appreciate it.

Reply

pinkpanther 11 November 2017 at 07:28

I had some question in my mind. I hope you would be able to help me.

Simply,

Product#AddToCart(Cart) (or Product#CreateLineItem(Cart))

(vs)
Cart#AddProduct(Product)

Which is more preferable?. I know DDD depends on the domain, but this use case is fairly popular. Hence my question.

Thanks in advance.

Reply

Replies

Zan Kavtaskin 20 November 2017 at 11:53

Hello,

It depends on the object that you are interacting with and the context that you are in. However, in a generic scenario I would prefer Cart.Add(Product). This is because when you are in the shop you pick up a cart and you add products in to the cart. You don't pick up a product and put cart in to it.

I hope this helps.

Reply

Anonymous 29 November 2017 at 12:44

Great post!

Have one question, regarding persistence using nhibernate.

In the sample code, there were Customer and CreditCard.
Customer have many CreditCard, CreditCard has Customer,
basically 1-to-M and M-to-1 nhibernate mapping respectively when using anemic models.

Based on my understanding of ddd, when creating a Customer, it should be loaded with all the info it needs, like the details and, based on the example, the "CreditCards" property.
If that property is exposed as ReadCollection and get only, how it is going to be populated?

Was thinking of these solution, but also the issues it will introduce
1. Add "credits" parameter in the static Create method
- but CreditCard need "customer" parameter in its own Create method
2. Add "AddCreditCard" method in Customer, to populate that readonly property
- but the Repository now need to know this detail, must ensure that it populated the credits property before returning an instance of a Customer.

Thanks!.

Reply

Unknown 30 November 2017 at 09:31

Great post. I have a question though - why would you use Create factory methods to create new instances instead of plain old constructors?

In this case we will have

Cart cart = new Cart(new List() { new Product(), new Product() });
Customer customer = new Customer("josh", "smith", "josh.smith@microsoft.com");
Purchase purchase = customer.Checkout(cart);

Reply

Replies

Zan Kavtaskin 7 February 2018 at 14:40

Hello Alexander,

It's partially preference, there are two reasons for this:
1. If constructor is used to create a cart, and constructor raises domain events then what should domain event be called? CartCreating or CartCreated? Technically Cart is not created until constructor has finished constructing the cart.

2. Constructors should be light and should just create the objects, even though domain event is a decoupled pub/sub pattern it's still synchronous and I really don't like the idea of my constructor executing some handles that are sending emails, auditing, etc.

Reply

Anonymous 25 February 2018 at 22:32

Another reason as to why static methods for Object creation is, Input Validation. If the requirement states that a customer object cannot be created with an invalid email. InvalidArgumentException is the only option with constructor approach. With static method approach, the failure reason can be sent back to the caller with some specialized return type that can hold success/failure flag, actual object and a proper error message in-case of a failure. Using exceptions to handle the control flow is bad and expensive too.

Reply

Anonymous 25 February 2018 at 22:43

Another reason could be, how the failed validation rules are sent to the caller. With constructor approach only option is to throw InvalidArgumentException. But, with static method you can communicate the object construction failure (because of valid validation rules, like invalid email for customer object creation from above code) using a specialized return type that can hold the actual object plus the failure reason in-case of a failure. This could be a better approach as using exceptions to alter the control flow is bad and expensive.

Reply

Unknown 10 September 2018 at 13:03

Hi Zan,

I have a question about moving from an anemic model to a domain driven design. In the anemic model, we were using data annotations which EF would use to map to tables in the database. With DDD, the models are not just data bags but contain the domain logic as well and are decoupled from persistence.

How does the infrastructure use this model with an ORM (EF is of interest to me)? How is the mapping to the database done? If the validation occurs in the domain, is there a duplicate logic going on in the infrastructure to setup (required, datalength, key, foreign key, and so forth) the DBContext? I would appreciate some insight here since I get the need to separate the model from persistence, but I don't see how you do this without repeating the model properties when setting up the database mappings.

Thanks

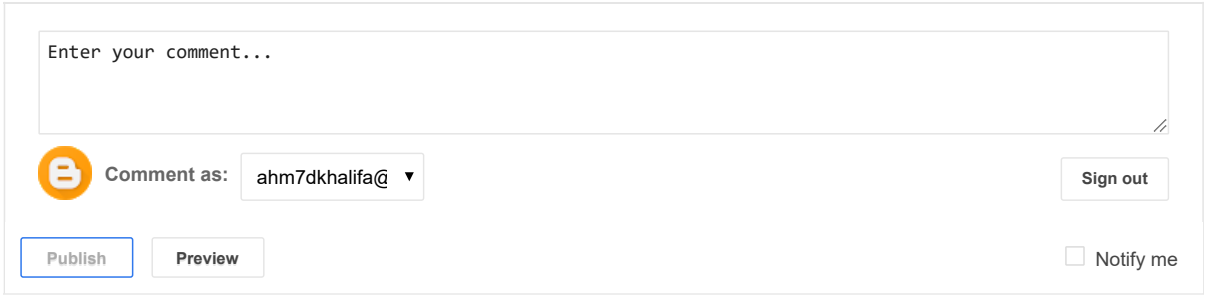
Reply

Nguyễn Thanh Tùng 15 October 2018 at 20:45

Hello Zan,
Thank you very much for your article. It's very awesome.
I would like to have a question about aggregate roots, the nested ones.
PRODUCT and CATEGORY are considered as aggregate roots, How can we organize models and repositories for them ?

Thank you

Reply



Newer Post Home

Subscribe to: Post Comments (Atom)