

ZAN KAVTASKIN

Musings about Software Engineering

Home	My Picks	Code Repositories	
------	----------	-------------------	--

Thursday, 26 September 2013

Applied Domain-Driven Design (DDD), Part 2 - Domain Events

In my last post we have addressed DDD thought process and constant refining/re-factoring. In this post we are going to talk about domain events. There are many articles on this out there (see references at the bottom), so i will be brief.

When something has happened in the domain, domain event can be raised. It can be from a trivial property change to an overall object state change. This is a fantastic way to describe an actual event in your domain, i.e. customer has checked out, customer was created, etc.

Let's extend our previous e-commerce example:

```
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();
```

Search This Blog

About Me



 **Zan Kavtaskin**

Nottingham, United Kingdom

I am a Software Director, Architect and Engineer. I work at MHR as a Software Delivery Director and I have also written software for companies such as Experian,

Emirates and Royal Mail.

[View my complete profile](#)

Popular Posts

[Applied Domain-Driven Design \(DDD\), Part 1 - Basics](#)

[Applied Domain-Driven Design \(DDD\), Part 0 - Requirements and Modelling](#)

[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 6 - Application Services](#)

[Applied Domain-Driven Design \(DDD\), Part 5 - Domain Service](#)

[Applied Domain-Driven Design \(DDD\), Part 7 - Read Model](#)

[Applied Domain-Driven Design \(DDD\) - Event Logging & Sourcing For Auditing](#)

[Unit Of Work Abstraction For NHibernate or Entity Framework C# Example](#)

```
        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);
        DomainEvents.Raise(new CustomerCheckedOut() { Purchase = purchase });
        return purchase;
    }

    public static Customer Create(string firstName, string lastName, string email)
```

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...](#)

```

    {
        Customer customer = new Customer()
        {
            FirstName = firstName,
            LastName = lastName,
            Email = email
        };
        return customer;
    }
}

public class CustomerCheckedOut : IDomainEvent
{
    public Purchase Purchase { get; set; }
}

public class CustomerCheckedOutHandle : Handles CustomerCheckedOut
{
    public CustomerCheckedOutHandle()
    {
    }

    public void Handle(CustomerCheckedOut args)
    {
        //send notifications, emails, update third party systems, etc
    }
}

```

For example in our case, when customer calls `Customer.Checkout(...)` we raise "CustomerCheckedOut" event. When event is handled it should not change the purchase object state, it should facilitate additional behavior only. For example sending out an email, updating financial monthly balance sheet, calling third party API, etc.

This is how you could use it:

```

public class CustomerCheckedOut : IDomainEvent
{
    public Purchase Purchase { get; set; }
}

public class CustomerCheckedOutHandle : Handles CustomerCheckedOut
{
    readonly IEmailSender emailSender;

    public CustomerCheckedOutHandle(IEmailSender emailSender)
    {
        this.emailSender = emailSender;
    }
}

```

```

    public void Handle(CustomerCheckedOut args)
    {
        this.emailSender.Send(args.Purchase.Customer, EmailTemplate.PurchaseMade);
        //send notifications, update third party systems, etc
    }
}

public interface IEmailSender
{
    void Send(Customer customer, EmailTemplate template);
}

public enum EmailTemplate
{
    PurchaseMade,
    CustomerCreated
}

```

It can be confusing, what do you put in to Customer.Checkout(...) method and what do you put in to the handler?

In my opinion Customer.Checkout(...) method should do what it needs to do with the access to the properties/fields that it has access to. So creating a purchase object and adding it to the collection, incrementing purchase count on the customer, updating last purchase date on the customer object, you get the idea.

What does handler have access to? All of the infrastructure layer interfaces. This makes it a great place to send out emails and notifications, synchronize with third party services, create audit records, etc.



Would like to see full working example?
Browse "Domain-Driven Design Example" Repository On Github

Summary:

- Domain event handlers should not change the state of the object that caused domain event to be triggered.
- Domain events should be raised inside the domain.
- Domain events handlers should be looked at as a side effect / chain reaction facilitator.

Useful links:

- [Domain Events - Salvation](#)
- [Strengthening your domain - domain events](#)

**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 (29 Votes)



Posted by Zan Kavtaskin

Labels: domain-driven design, software engineering

10 comments:

**Pavan** 30 September 2016 at 03:15

Dear Zan,

The architecture was cool and complete, But if i want to wrap unit of work for each entity, How can I do this? Also what is the difference between having Service wrapper layer in between the API and Repository and Domain Services? Please clarify.

Thanks,
Pavan

[Reply](#)

▼ Replies

**Zan Kavtaskin** 9 October 2016 at 05:16

Thank you for reading and commenting!

Q: How can you wrap each entity in unit of work?

Approach 1:

Make your services more fine grained so that you are working only with one entity.

You can find example of this in here (see "Remove" method) : <https://github.com/zkavtaskin/Domain-Driven-Design-Example/blob/master/eCommerce/ApplicationLayer/Carts/CartService.cs>

Approach 2:

If approach 1 is not possible take control of the Unit Of Work flow, here is pseudocode example:

```
MyService(IUnitOfWork, IRepositoryA, IRepositoryB) {  
    SomeBusinessOperation(someValue) {
```

```
        IUnitOfWork.BeginTransaction()  
        ListA As = IRepositoryA.GetAll();  
        for(A a in As) {  
            A.someValue = someValue;  
        }  
        IUnitOfWork.Commit();
```

```
        IUnitOfWork.BeginTransaction()  
        ListB Bs = IRepositoryB.GetAll();  
        for(B b in Bs) {
```

```

B.someValue = someValue;
}
IUnitOfWork.Commit();

}
}

```

Q: What's the difference between Application Services and Domain Services?

Application Service is there to control the Unit Of Work and it's there to get the Entity from the repository. Application Service should not contain any business logic. It should be super thin.

Domain Service lives inside your domain model. This is where you can put your entity orchestration business logic. Application Service and Domain Events can call your Domain Service. Domain Service allows you to capture logic that doesn't belong in the Domain Entity or Application Service and Domain Service allows you to orchestrate between different Domain Entities.

Here is an example:

<https://github.com/zkavtaskin/Domain-Driven-Design-Example/blob/master/eCommerce/DomainModelLayer/Services/CheckoutService.cs>

Pavan, I hope this helps!

[Reply](#)



Pavan 9 October 2016 at 05:26

Dear Zan,

Many thanks for clarifying my doubts.. Your answers gave me a clarity on implementing UOW.I will share the pseudo code once i'm done with it.

Thank you again for giving such a wonderful seamless architecture. :)

[Reply](#)

▼ Replies



Zan Kavtaskin 27 July 2017 at 15:55

Thank you for the kind words!

[Reply](#)



Pavan 13 January 2017 at 02:24

Dear Zan,

As you said, Approach 1 looks not promising as i need to eye on every entity. In the service wrapper layer i'm using the Approach 2 which is an abstraction of UOW and NHibernate, Simply wrapping my controller methods with WebApi annotation worked for me. Like below:

```

[WebApiUnitOfWork(IsolationLevel.ReadCommitted)]
public HttpResponseMessage Get()

```

Also can we put fetching logic alone in the service wrapper layer like below? I'm having a reference to the corresponding repository of the entity like below.

```

public User Register(User user)
{
    bool isLoginAvailable = Repository.Any(x => x.Email.Equals(user.Email)) == false;

```

```
//Other implementation and return user object logic  
}
```

Thanks,
Pavan

Thanks,
Pavan

[Reply](#)

▼ Replies



Zan Kavtaskin 27 July 2017 at 16:03

Hello Pavan,

I would not expose IsolationLevel in the controller, isolation level is an infrastructure concern.

I don't see anything wrong with Application Layer doing something like this:
`bool isLoginAvailable = Repository.Any(x => x.Email.Equals(user.Email)) == false;`

Thanks for sharing!

[Reply](#)



Unknown 25 January 2018 at 18:04

Dear Zan

If a domain entity or a domain server can be a event handle?

[Reply](#)

▼ Replies



Zan Kavtaskin 2 February 2018 at 08:37

Hello Joey,

Can you please clarify / expand your question?

[Reply](#)



Lucas 15 March 2018 at 06:54

What do you think about this architecture?
<https://github.com/EduardoPires/EquinoxProject>

The application layer publish notifications using MediatR. These commands are handled in the domain layer where it can be persisted. Then a new notification has published depending of status of the command handling(success, failure).

[Reply](#)




Lucas 15 March 2018 at 06:54

This comment has been removed by the author.

[Reply](#)

Enter your comment...

 Comment as: ahm7dkhalifa@ ▼ Sign out

Publish Preview ☐ Notify me

[Newer Post](#) [Home](#) [Older Post](#)Subscribe to: [Post Comments \(Atom\)](#)

© Zan Kavtaskin. Powered by [Blogger](#).