SOFTWARE ENGINEERING

Find the DDD Aggregate Root

Asked 2 years, 8 months ago Active 1 year, 8 months ago Viewed 10k times

10

13

Let's play everyone's favorite game, find the Aggregrate Root. Let's use the canonical Customer/Order/OrderLines/Product problem domain. Traditionally, Customer, order, and product are the AR's with OrderLines being entities under the Order. The logic behind this is that you need to identify customers, orders, and products, but an OrderLine wouldn't exist without an Order. So, in our problem domain, we have a business rule saying that a Customer can only have one undelivered order at a time.

Does that move the order under the customer aggregate root? I think it does. But in doing so, that makes the Customer AR rather large and subject to concurrency issues later.

Or, what if we had a business rule stating that a customer can only order a particular product once in its lifetime. This is more evidence requiring the Customer to own the Order.

But when it comes to shipping, they do all of their actions on the Order, not the customer. It's kind of dumb to have to load up the entire customer in order to mark an individual Order as delivered.

This is what I'm proposing:

```
class Customer
{
    public Guid Id {get;set;}
```

```
    public string Name { get; set; }
    public Address Address { get; set; }
    public IEnumerable<Order> Orders { get; set; }
    public void PlaceOrder(ThingsInTheOrder thingsInTheOrder)
    {
        // Make sure there aren't any pending orders already.
        // Make sure they aren't ordering a Widget if they've already ordered a Widget
in the past.
        // Create an Order object and add it to the collection.  Raise a domain event to
trigger emails and other stuff
    }
}

class Order
{
    public Guid Id { get; set; }
    public IEnumerable<OrderLine> OrderLines { get; set; }
    public ShippingData {get;set;}
    public void Ship(ShippedByPerson shippedByPerson, string trackingCode)
    {
         // Create a new ShippingData object and assign it from the data passed in.
         // Publish a domain event
    }
}
```

My biggest concern is the concurrency issue and the fact that the Order itself has characteristics of an aggregate root.

domain-driven-design

asked Aug 14 '17 at 17:06

Darthg8r

207 ● 2 ● 5

5 Answers

Active | Oldest | Votes

12

Short Version

The rationale for DDD is that Domain Objects are abstractions which should fulfil your functional domain requirements - if Domain Objects are unable to easily fulfil those requirements, it suggests you might be using the wrong abstraction.

Naming *Domain Objects* using *Entity Nouns* can lead toward those objects becoming tightly-coupled with each other and becoming bloated "god" objects, and they can throw up issues such as the one in this question such as "Where is the right place to put the CreateOrder method?".

To make it easier to identify the 'right' Aggregate Root, consider a different approach where Domain Objects are based on the functional high-level business requirements - i.e. choose nouns which allude to functional requirements and/or behaviours that users of the system need to perform.

Long Version

DDD is an approach to OO Design which is intended to result in a graph of *Domain Objects* in the *Business Layer* of your system - Domain objects are responsible for satisfying your High-level Business requirements, and ideally should be able to rely on the Data Layer for things like the performance and integrity of the underlying persistent data store.

Another way to look at it could be the bullet points in this list

- Entity Nouns typically suggest data attributes.
- Domain Nouns should suggest behaviour
- DDD and OO Modelling are concerned with abstractions based on functional requirements and core domain/business logic.
- The Business Logic Layer is responsible for satisfying high-level domain requirements

One of the common misconceptions regarding DDD is that Domain Objects should be based on some physical real-world "thing" (i.e. some noun which you can point to in the real world, attributed with all kinds of data/properties), however the data/attributes of those real world things do not necessarily make a good starting point when trying to nail down functional requirements.

Of course, Business Logic should *use* this data, but Domain Objects themselves should ultimately be abstractions which represent functional Domain requirements and behaviours.

For example; nouns such as `Order` or `Customer` do not imply any behaviour, and therefore are generally unhelpful abstractions for representing business logic and Domain Objects.

When looking for the kinds of abstractions which might be useful for representing Business Logic, consider typical requirements which you might expect a system to fulfil:

- As a Salesperson, I want to Create an Order for a New Customer so that I can generate an invoice for the Products to be sold with their Prices and Quantity.
- As a Customer Service Advisor I want to Cancel a Pending Order so that the Order is not fulfilled by a Warehouse Operator.

- As a Customer Service Advisor I want to Return an Order Line so that the Product can be adjusted into the Inventory and Payment is Refunded back via the Customer's original Payment method.
- As a Warehouse Operator I want to view all Products on a Pending Order and the Shipping information so that I can Pick the products and ship them via the Courier.
- etc.

Modelling Domain Requirements with a DDD Approach

Based on the above list, consider some potential Domain Objects for such an Orders system:

```
SalesOrderCheckout
PendingOrdersStream
WarehouseOrderDespatcher
OrderRefundProcessor
```

As domain objects, these represent abstractions which take ownership of various behavioural domain requirements; indeed their nouns hint strongly at the specific functional requirement(s) they fulfil.

(There may be be additional infrastructure in there too such as an `EventMediator` to pass notifications for observers wanting to know when a new order has been created, or when an order has been shipped, etc).

For example, `SalesOrderCheckout` probably needs to handle data about Customers, Shipping and Products, however isn't concerned with anything to do with the behaviour for shipping orders, sorting pending orders or issuing refunds.

For `SalesOrderCheckout` to fulfil its domain requirements includes enforcing those business rules such as preventing customers ordering too many items, possibly running some validation, and perhaps raising notifications for other parts of the system - it can do all of those things without necessarily needing to depend on any of the other objects.

DDD using Entity Nouns to represent Domain Objects

There are a number of potential dangers when treating simple nouns such as `Order`, `Customer` and `Product` as Domain Objects; among those problems are those you allude to in the question:

- If a method handles an `Order`, a `Customer` and a `Product`, which Domain Object does it belong to?
- Where is the Aggregate Root for those 3 Objects?

If you choose Entity Nouns to represent Domain Objects, a number of things may happen:

- `Order`, `Customer` and `Product` are at risk of growing into "god" objects

- Risk of ending up with a single `Manager` god-object to tie everything together.
- Those objects risk becoming tightly coupled to each other - it may be hard to fulfil domain requirements without passing `this` (or `self`)
- A risk of developing "leaky" abstractions - i.e. domain objects being expected to expose dozens of `get`/`set` methods which weaken encapsulation (or, if you don't, then some other programmer probably will later on..).
- A risk of Domain Objects becoming bloated with a complex mixture of business data (e.g. user data input via a UI) and transitory state (e.g. a 'history' of user actions when the order has been modified).

DDD, OO Design and Plain Models

A common misconception regarding DDD and OO Design is that "plain" models are somehow 'bad' or an 'anti-pattern'. Martin Fowler wrote an article describing the Anaemic Domain Model - but as he makes it clear in the article, DDD itself should not 'contradict' the approach of clean separation between layers

> *"It's also worth emphasizing that putting behavior into the domain objects should not contradict the solid approach of using layering to separate domain logic from such things as persistence and presentation responsibilities. The logic that should be in a domain object is domain logic - validations, calculations, business rules - whatever you like to call it."*

In other words, using plain Models for holding business data transferred between other layers (e.g. an Order model passed in by a user application when the user wants to create a new order) is not the same thing as an "Anaemic Domain Model". 'plain' data models are often the best way to track data and transfer data between layers (such as a REST web service, a persistence store, An application or UI, etc).

Business logic may process the data in those models and may track them as part of the business state - but won't necessarily take ownership of those models.

The Aggregate Root

Looking again at the example Domain Objects - `SalesOrderCheckout`, `PendingOrdersStream`, `WarehouseOrderDespatcher`, `OrderRefundProcessor` there's still no obvious Aggregate Root; but that doesn't actually matter because the these Domain Objects have wildly separate responsibilities which don't seem to overlap.

Functionally, there's no need for the `SalesOrderCheckout` to talk to the `PendingOrdersStream` because the job of the former is complete when it has added a new order to the Database; on the other hand, the `PendingOrdersStream` can retrieve new orders from the Database. These objects don't actually need to interact with each other directly (Perhaps an Event Mediator might provide notifications between the two, but I would expect any coupling between these objects to be very loose)

Perhaps the Aggregate Root will be an IoC Container which injects one or more of those Domain Objects into a UI Controller, also supplying other infrastructure like `EventMediator` and `Repository`. Or perhaps it will be some kind of lightweight Orchestrator Service sitting on top of the Business Layer.

The Aggregate root doesn't *necessarily* need to be a Domain Object. For the sake of keeping Separation of Concerns between Domain objects, it's generally a good thing when the aggregate root is a separate object with no business logic.

edited Aug 15 '17 at 7:39

answered Aug 14 '17 at 21:16

Ben Cottrell

8,123 ● 3 ● 22 ● 30

3 I downvoted because your answer conflates concepts from Entity Framework which is a Microsoft-specific technology with Domain Driven Design, which is from a book written by Eric Evans of the same name. You have some statements in your answer that are in direct contradiction with the DDD book and this question makes zero mention of Entity Framework but specifically is tagged with DDD. There's also no mention of persistence at all in the question so I don't see where database tables are relevant. – RibaldEddie Aug 14 '17 at 22:34

@RibaldEddie Thanks for taking the time to review the answer and comment, I agree the mention of persistent data doesn't really need to be in the answer so I've reworded it to remove that. The main focus of the answer could be summarised as "Entity Nouns often aren't very good Domain Object class names due to their tendency to become tightly-coupled bloated god objects", hopefully the message and reasoning WRT functional requirements/behaviour is clearer now? – Ben Cottrell Aug 15 '17 at 7:27

The DDD book doesn't have that concept IIRC. It has Entities, which are merely classes that when instantiated have a persistent and unique identity so that two separates instances imply two unique and persist-able things, which contrasts with Value Objects which don't have any identity and don't persist over time. In the book, both Entities and Value objects are domain objects. – RibaldEddie Aug 15 '17 at 16:03

What's the criteria for defining an aggregate ?

10

Let's go back to the basics of the big blue book:

> **Aggregate:** A cluster of associated objects that are ***treated as a unit for the purpose of data changes***. External references are restricted to one member of the AGGREGATE, designated as the root. A set of consistency rules applies within the AGGREGATE'S boundaries.

The goal is to maintain the invariants. But it's also to manage properly local identity, i.e. the identify of objects which do not have a meaning alone.

`Order` and `Order line` definitively belong to such a cluster. For example:

- Delete an `Order`, will require deletion of all its lines.
- Deleting a line might require renumbering of the following lines
- Adding a new line would require to determine the line nulber based on all the other lines of the same order.
- Changing some order information, such as for example the currency, might affect the meaning of the price in the line items (or require to recalculate the prices).

So here the full aggregate is required to ensure consistency rules and invariants.

When to stop ?

Now, you describe some business rules, and argue that to ensure them, you'd need to consider the customer as part of the aggregate:

> We have a business rule saying that a Customer can only have one undelivered order at a time.

Of course, why not. Let's see the implications: the order would always be accessed via the customer. Is this real life ? When workers are filling the boxes for delivering the order, will they need to read the customer bar code and the order barcode to access the order ? In fact, in general, the identity of an Order is global not local to a customer, and this relative independence suggests to keep him outside the aggregate.

In addition, these business rules look more as policies: it's an arbitrary decision of the company to run their process with these rules. If the rules are not respected, the boss might be unhappy, but the data is not really inconsistent. And moreover, overnight "per customer one undelivered order at a time" could become "ten undelivered orders per customer" or even "independently of the customer, hundred undelivered orders per warehouse", so that the aggregate might no longer be justified.

answered Aug 14 '17 at 21:56

Christophe

40.5k ● 5 ● 60 ● 99

1

> in our problem domain, we have a business rule saying that a Customer can only have one undelivered order at a time.

Before you go too deep down that rabbit hole, you should review Greg Young's discussion of set consistency, and in particular:

> *What is the business impact of having a failure?*

Because in a lot of cases, the right answer isn't to try to prevent the wrong thing from happening, but instead to generate exception reports when there might be a problem.

But, presuming that multiple undelivered orders are a significant liability to your business....

Yes, if you want to be ensuring that there is only one undelivered order, then there needs to be some aggregate that can see all of the orders for a customer.

That aggregate is not necessarily the customer aggregate.

It might be something like an order queue, or an order history, where all of the orders for a specific customer go into the same queue. From what you have said, it doesn't need all of the customer's profile data, so that shouldn't be part of this aggregate.

> But when it comes to shipping, they do all of their actions on the Order, not the customer.

Yes, when you are actually working with fulfillment and pull sheets, the history view isn't particularly relevant.

The history view, to enforce your invariant, only needs the order id and its current processing status. That doesn't necessarily need to be part of the same aggregate as the order -- remember, aggregate boundaries are about managing change, not structuring views.

So it could be that you handle the order as an aggregate, and the order history as a separate aggregate, and coordinate the activity between the two.

answered Aug 15 '17 at 2:49

VoiceOfUnreason

22.7k ● 1 ● 30 ● 54

1

You've set up a straw person example. It's too simplistic and I doubt it reflects a real-world system. I wouldn't model those Entities and their related behavior in the way that you specified because of that.

Your classes need to model the state of an order in a way that is reflected in multiple aggregates. For example when the customer puts the system into the state where the customer's order request needs to be processed, I might create a domain entity object aggregate called `CustomerOrderRequest` or `PendingCustomerOrder` or even just `CustomerOrder`, or whatever language the business uses,

and it could hold a pointer to both the Customer and the OrderLines and then have a method like `canCustomerCompleteOrder()` which is called from the service layer.

This domain object would contain the business logic to determine whether or not the order was valid.

If the order were valid and processed then I'd have some way to transition this object to another object that represented the processed order.

I think the problem with your understanding is that you're using a contrived over-simplified example of aggregates. A `PendingOrder` can be it's own aggregate separate from an `UndeliveredOrder` and again separate from a `DeliveredOrder` or a `CancelledOrder` or whatever.

edited Aug 15 '17 at 3:20

answered Aug 14 '17 at 17:53

RibaldEddie

2,765 ● 1 ● 10 ● 15

Although your attempt at gender-neutral language is amusing, I would note that women never stand in fields to scare off crows. – Robert Harvey Aug 14 '17 at 19:23

@RobertHarvey that's a strange thing to focus on in my post. Scarecrows and effigies both regularly have appeared in female form throughout history. – RibaldEddie Aug 14 '17 at 19:30

You wouldn't have made the distinction in your post if you didn't consider it important. As a matter of linguistics, the term is "straw man;" any reservations about sexism are almost certainly trumped by the "what in the hell is he talking about" factor created by inventing your own term. – Robert Harvey Aug 14 '17 at 19:35

5 @RobertHarvey if someone knows what straw man means, I'm sure they can figure out what straw person means if they haven't heard that term. Can we focus on the substance of my post please wrt software? – RibaldEddie Aug 14 '17 at 19:37

Vaughn Vernon mentions this in his book "Implementing Domain-Driven Design" at the beginning of Chapter 7 (Services):

1

"Often the best indication that you should create a Service in the domain model is when the operation you need to perform feels out of place as a method on an Aggregate or a Value Object".

So, in this case there could be a domain service called "CreateOrderService" which takes a Customer instance and the list of items for the order.

```
class CreateOrderService
{
    public Order CreateOrder(Customer customer, ThingsInTheOrder thingsInTheOrder)
    {
        // Get all the orders for the customer
        // Check if any of the things to be ordered exist in previous orders
        // If none have been previously ordered, create the order and return it
        // Otherwise return null
    }
}
```

answered Aug 26 '18 at 18:06

claudius

11 ● 1

1 Can you please explain more how domain service can help on addressing concurrency concern in the question? – ivenxu Aug 27 '18 at 0:36