DDD - Modifications of child objects within aggregate

Asked 7 years, 10 months ago Active 2 months ago Viewed 4k times

12

5

I am having some difficulty working out the best way to handle a fairly complex scenario. I've seen quite a few similar questions, but none addressed this scenario to my satisfaction.

An Order (aggregate root) is created with multiple OrderLines (child entities). According to business rules, each OrderLine must maintain the same identity for the life of the Order. OrderLines have many (20+) properties and can be mutated fairly often before the Order is considered "locked". In addition, there are invariants that must be enforced at the root level; for example, each Order Line has a quantity and the total quantity for the Order cannot exceed X.

I'm not sure how to model this scenario when considering changes to OrderLines. I have 4 choices that I can conceive of, but none seem satisfactory:

1) When the time comes to modify an OrderLine, do it using a reference provided by the root. But I lose the ability to check invariant logic in the root.

```
var orderLine = order.GetOrderLine(id);
orderLine.Quantity = 6;
```

2) Call a method on the order. I can apply all invariant logic, but then I'm stuck with a proliferation of methods to modify the many properties of the OrderLine:

```
order.UpdateOrderLineQuantity(id, 6);
order.UpdateOrderLineDescription(id, description);
order.UpdateOrderLineProduct(id, product);
...
```

3) This might be easier if I treated the OrderLine as a Value Object, but it has to maintain the same identity per business requirements.

4) I can grab references to the OrderLines for modifications that do not affect invariants, and go through the Order for those that do. But then what if invariants are affected by most of the OrderLine properties? This objection is hypothetical, since only a few properties can affect invariants, but thhat can change as we uncover more business logic.

Any suggestions are appreciated...do not hesitate to let me know if I'm being dense.

domain-driven-design aggregateroot

asked May 23 '12 at 21:32

Cork
551 ● 5 ● 15

6 Answers

Active | Oldest | Votes

5

1. Is not optimal because it allows breaking domain invariant.
2. Will result in code duplication and unnecessary method explosion.
3. Is the same as 1). Using Value Object will not help with maintaining domain invariant.
4. This option is what I would go with. I would also not worry about potential and hypothetical changes until they materialize. The design will evolve with your understanding of the domain and can always be refactor later. There is really no value in hindering your existing design for the sake of some future changes that may not happen.

answered May 23 '12 at 23:17

Dmitry
15.4k ● 2 ● 36 ● 63

Thank you for your answer - I think it is probably my best choice of the ones I presented. I was actually hoping for someone to suggest a better pattern I may have overlooked ;) I am a little leery of using it, because it's not clean. Or maybe a better way to put it...not consistent, as @eulerfx

pointed out. But I guess it will do for now... – Cork May 24 '12 at 14:14

2 I know I accepted this as an answer already...but I thought of a different approach. Would it be OK to to have a Save(IOrderLine) method on the Order? Then I can pass a ref to the Order, avoid a bunch of granular methods, and still allow the Order to enforce invariants. – Cork Jun 8 '12 at 18:55

5

One drawback of 4 as opposed to 2 is lack of consistency. In certain instances it might be beneficial to maintain a degree of consistency in regards to updating order line items. It may not be immediately clear why certain updates are done through the order while others through the order line item. Furthermore, if order lines have 20+ properties, perhaps that is a sign that there is potential for grouping among those properties resulting in fewer properties on the order line. Overall, approach 2 or 4 is fine as long as you make sure to keep operations atomic, consistent and in correspondence with ubiquitous language.

answered May 24 '12 at 2:10

eulerfx

31.8k ● 5 ● 56 ● 77

1 Thanks for the answer! I agree with the lack of consistency, but I think it's my best option. We actually started off with a lot of the Order Line properties sitting in the Order, but as we continued to analyze the domain we found the Order Line was a more suitable spot. It is possible, however, that we got carried away a little... – Cork May 24 '12 at 14:18

5

There is a fifth way of doing this. You can fire a domain event (e.g. `QuantityUpdatedEvent(order, product, amount)`). Let the aggregate handle it internally by going through the list of orderlines, select the one with matching product and update its quantity (or delegate the operation to `OrderLine` which is even better)

answered Jul 31 '12 at 10:22

Jeroen

879 ● 1 ● 6 ● 16

1 When an entity has children that can be modified in many different ways, I agree that domain events are usually the right way to go. Jimmy Bogard has also written some nice pieces here and here. – Justin J Stark Feb 26 '15 at 16:53

4

The domain event is the most robust solution.

However if that's overkill you can also do a variation of #2 using the Parameter Object pattern - have a single ModfiyOrderItem function on the entity root. Submit a new, updated order item, and then internally the Order validates this new object and makes the updates.

So your typical workflow would turn into something like

```
var orderItemToModify = order.GetOrderItem(id);
orderItemToModify.Quantity = newQuant;

var result = order.ModifyOrderItem(orderItemToModfiy);
if(result == SUCCESS)
{
  //good
 }
else
{
   var reason = result.Message; etc
}
```

The main shortcoming here is it allows a programmer to modify the item, but not commit it and not realize that. However it is easily extensible and testable.

answered Feb 14 '13 at 17:47

Mathieson

1,389 ● 1 ● 10 ● 16

1 This scenario is covered in DDD Quickly. Which states `It is possible for the root to pass transient references of internal objects to external ones, with the condition that the external objects do not hold the reference after the operation is finished. One simple way to do that is to pass copies of the Value Objects to external objects.` Which I interpret as passing a DTO to the external object for modification, and accepting that DTO to apply modifications. Alternatively, you could have methods on your root that modify the child. Such as `order.ChangeItemQuantity(id, quantity)` – Douglas Gaskell Jul 19 '19 at 20:09

2

Here's another option if your project is small and you want to avoid the complexity of domain events. Create a service that handles the rules for Order and pass it in to the method on OrderLine:

```
public void UpdateQuantity(int quantity, IOrderValidator orderValidator)
{
    if(orderValidator.CanUpdateQuantity(this, quantity))
        Quantity = quantity;
}
```

CanUpdateQuantity takes the current OrderLine and the new quantity as arguments. It should lookup the Order and determine if the update causes a violation in the total Order quantity. (You will have to determine how you want to handle an update violation.)

This may be a good solution if your project is small and you don't need the complexity of domain events.

A downside to this technique is you are passing a validation service for Order into OrderLine, where it really doesn't belong. By contrast, raising a domain event moves the Order logic out of OrderLine. The OrderLine can then just say to the world, "Hey, I'm changing my quantity." and the Order validation logic can take place in a handler.

edited Feb 26 '15 at 21:46

answered Feb 26 '15 at 21:41

Justin J Stark

416 ● 7 ● 14

0

what about using DTO ?

```
public class OrderLineDto
{
    public int Quantity { get; set; }
    public string Description { get; set; }
    public int ProductId { get; set; }
}

public class Order
{
    public int? Id { get; private set; }
    public IList<OrderLine> OrderLines { get; private set; }

    public void UpdateOrderLine(int id, OrderLineDto values)
    {
        var orderLine = OrderLines
            .Where(x => x.Id == id)
            .FirstOrDefault();

        if (orderLine == null)
        {
            throw new InvalidOperationException("OrderLine not found.");
        }

        // Some domain validation here
        // throw new InvalidOperationException("OrderLine updation is not valid.");

        orderLine.Quantity = values.Quantity;
```

```
        orderLine.Description = values.Description;
        orderLine.ProductId = values.ProductId;
    }
}
```

- only issue here is that `OrderLines` property has public getter and user of this class can add item into collection. The only way I can think of how to prevent it, is to hide getter and add new getter which will return collection of DTO's if there is a need for that.
- the id parameter of `UpdateOrderLine` method can be part of DTO, it would probably work better with that
- probably you can accept `OrderLine` as a parameter directly instead of `OrderLineDto` (if you would like to use some `OrderLine` validation before passing to `Order`).

edited Feb 8 at 22:49

answered Feb 8 at 22:28

Muflix

3,431 ● 6 ● 39 ● 94