Validation and DDD

September 13, 2016

Validation and DDD can be a tricky combination. How to perform validation in a way that doesn't lead to domain knowledge leakage?

1. Validation and DDD

Recently, I came across an interesting discussion on the Jimmy Bogard's blog. The post itself is quite old but the subject is still relevant. The discussion was about where to implement validation: in aggregates or in application services. It has resonated with me and I'd like to write where I personally stand on this matter.

If you look closely at situations where you need to validate something, most of them will fall into two categories: task-based validations and CRUD-y validations. Let's start with the task-based one.

Suppose you have an Order class which looks like this:

```
public class Order
{
    public OrderStatus Status { get; private set; } = OrderStatus.InProgress;
    public string DeliveryAddress { get; private set; }
    public DateTime DeliveryTime { get; private set; }
}

public enum OrderStatus
{
    InProgress,
    Delivering,
    Closed
}
```

Let's say that you need to implement a delivery feature. There are two business rules that are attached to that feature: all orders must have a non-empty delivery address and the day of delivery cannot be Sunday.

There are several common ways of enforcing these business rules.

2. Solution 1: IsValid method

The first method would be to assign an address and time to an order and make it validate itself. The validation method should then return any errors it encountered back to the caller. The Order class would look like this:

Vladimir Khorikov

```
public class Order
{
    public OrderStatus Status { get; private set; } = OrderStatus.InProgress;
    public string DeliveryAddress { get; set; }
    public DateTime DeliveryTime { get; set; }

    public IReadOnlyList<string> ValidateForDelivery()
    {
        var errors = new List<string>();

        if (string.IsNullOrWhiteSpace(DeliveryAddress))
            errors.Add("Must specify delivery address");

        if (DeliveryTime.DayOfWeek == DayOfWeek.Sunday)
            errors.Add("Cannot deliver on Sundays");

        return errors;
    }

    public void Deliver()
    {
        Status = OrderStatus.Delivering;
    }
}
```

And here's the code from an app service that uses it:

```
public class OrderController
{
    public ActionResult Deliver(DeliveryViewModel model)
    {
        Order order = GetFromDatabase(model.OrderId);
        order.DeliveryAddress = model.Address;
        order.DeliveryTime = model.Time;

        IReadOnlyList<string> errors = order.ValidateForDelivery();
        if (errors.Any())
        {
            ModelState.AddModelError("", string.Join(", ", errors));
            return View();
        }

        order.Deliver();
        // Save the order and redirect to a success page
    }
}
```

The benefit of this approach is that it allows you to concentrate the relevant domain logic within the aggregate thus preventing its leakage. The knowledge regarding what it means to be ready for delivery should clearly belong to the Order class itself as that knowledge fully depends on the data that resides inside that class.

There's a drawback to this implementation too, and it's quite substantial. In order to validate itself, the entity must enter an invalid state first. And that means the aggregate no longer maintains its consistency boundary. Its invariants are held loosely and can be broken at any time.

This drawback is a deal-breaker. Invariants that lie within an aggregate should be kept by that aggregate at all times. Their violation leads to many nasty situations, potential data corruption is one of them.

3. Solution 2: checking validity in the application services layer

Another way to implement this task is to check the incoming request in the application service and send the data to the entity only if it passes the validation. You can do that either manually or with some automatic tool, such as ASP.NET model binder or FluentValidator.

– Entity vs Value Object: the ultimate list of differences
– DTO vs Value Object vs POCO
– 3 misuses of ?. operator in C# 6
– Specification pattern: C# implementation
– Database versioning best practices
– Unit testing private methods
– Functional C#: Handling failures, input errors
– REST API response codes: 400 vs 500

RECENT ARTICLES
– Validation and DDD: New online course
– Static methods considered evil?
– Always-Valid Domain Model
– How to handle unique constraint violations
– C# 9 Records as DDD Value Objects
– Domain model purity and lazy loading
– Domain model purity and the current time
– Domain model purity vs. domain model completeness
– Is Equality Comparison a Code Pollution?
– How to Unit Test an Abstract Class

» All articles

Here's the code:

```
public class Order
{
    public OrderStatus Status { get; private set; } = OrderStatus.InProgress;
    public string DeliveryAddress { get; private set; }
    public DateTime DeliveryTime { get; private set; }

    public void Deliver(string address, DateTime time)
    {
        DeliveryAddress = address;
        DeliveryTime = time;
        Status = OrderStatus.Delivering;
    }
}

public class OrderController
{
    public ActionResult Deliver(DeliveryViewModel model)
    {
        Order order = GetFromDatabase(model.OrderId);

        // ModelState uses rules defined by FluentValidator
        // (or similar tool) to validate DeliveryViewModel
        if (!ModelState.IsValid)
        {
            return View();
        }

        order.Deliver(model.Address, model.Time);
        // Save the order and redirect to a success page
    }
}
```

What makes this approach so appealing is the declarative nature of the validation rules and the ability to attach them to ASP.NET's execution pipeline. Here they are defined using FluentValidator:

```
public class DeliveryViewModelValidator : AbstractValidator<DeliveryViewModel>
{
    public DeliveryViewModelValidator()
    {
        RuleFor(x => x.Address).NotEmpty().WithMessage("Must specify delivery address");
        RuleFor(x => x.Time).Must(NotBeSunday).WithMessage("Cannot deliver on Sundays");
    }

    private bool NotBeSunday(DateTime dateTime)
    {
        return dateTime.DayOfWeek != DayOfWeek.Sunday;
    }
}
```

The shortcoming of this approach is that it encourages business logic leakage. The Order entity is no longer accountable for holding the domain knowledge about its invariants. This responsibility is drifted away from the domain model to the application services layer.

Also, while we do check for validity of the entity prior to calling the `Deliver` method, there's nothing preventing us from assigning the entity incorrect data. Unlike in the previous code sample, we don't actively do that but this scenario remains possible. The aggregate in this implementation still doesn't maintain its invariants.

This implementation is still better than the previous one, though, and it might be fine to use it in simple code bases. In complex projects, however, it's important to keep the domain model invariants intact for maintainability reasons.

4. Solution 3: TryExecute pattern

The third way is to make the `Deliver` method do all the validations needed and either proceed with the delivery or return back any errors it encounters. Here's how it can be done:

```
public class Order
{
    public OrderStatus Status { get; private set; } = OrderStatus.InProgress;
    public string DeliveryAddress { get; private set; }
    public DateTime DeliveryTime { get; private set; }

    public IReadOnlyList<string> Deliver(string address, DateTime time)
    {
        var errors = new List<string>();

        if (string.IsNullOrWhiteSpace(address))
            errors.Add("Must specify delivery address");

        if (time.DayOfWeek == DayOfWeek.Sunday)
            errors.Add("Cannot deliver on Sundays");

        if (errors.Any())
            return errors;

        DeliveryAddress = address;
        DeliveryTime = time;
        Status = OrderStatus.Delivering;

        return errors;
    }
}

public class OrderController
{
    public ActionResult Deliver(DeliveryViewModel model)
    {
        Order order = GetFromDatabase(model.OrderId);

        IReadOnlyList<string> errors = order.Deliver(model.Address, model.Time);
        if (errors.Any())
        {
            ModelState.AddModelError("", string.Join(", ", errors));
            return View();
        }

        // Save the order and redirect to a success page
    }
}
```

This is essentially what `int.TryParse("5", out val)` does when you try to parse a value, hence the name TryExecute.

From the domain model purity standpoint, this approach is much better. The entity here both holds the domain knowledge and maintains its consistency. It's impossible to transition it into an invalid state, the invariants are guaranteed to be preserved.

5. Solution 4: Execute / CanExecute pattern

The implementation above violates the CQS principle: it both mutates the entity's state and returns a value. It's not a big deal but it would be nice to eliminate this shortcoming.

To do that, you can separate the `Deliver` method in two: `Deliver` and `CanDeliver`:

```
public class Order
{
    public OrderStatus Status { get; private set; } = OrderStatus.InProgress;
    public string DeliveryAddress { get; private set; }
    public DateTime DeliveryTime { get; private set; }

    public IReadOnlyList<string> CanDeliver(string address, DateTime time)
    {
        var errors = new List<string>();

        if (string.IsNullOrWhiteSpace(address))
            errors.Add("Must specify delivery address");

        if (time.DayOfWeek == DayOfWeek.Sunday)
            errors.Add("Cannot deliver on Sundays");

        return errors;
    }

    public void Deliver(string address, DateTime time)
    {
        if (CanDeliver(address, time).Any())
            throw new InvalidOperationException();

        DeliveryAddress = address;
        DeliveryTime = time;
        Status = OrderStatus.Delivering;
    }
}

public class OrderController
{
    public ActionResult Deliver(DeliveryViewModel model)
    {
        Order order = GetFromDatabase(model.OrderId);

        IReadOnlyList<string> errors = order.CanDeliver(model.Address, model.Time);
        if (errors.Any())
        {
            ModelState.AddModelError("", string.Join(", ", errors));
            return View();
        }

        order.Deliver(model.Address, model.Time);
        // Save the order and redirect to a success page
    }
}
```

This approach offers better separation of concerns due to adherence to CQS. Note that it's still impossible to transition the entity into an invalid state. The `Deliver` method declares a precondition saying that `CanDeliver` must hold true prior to executing the delivery. (Here you can read more on the subject of validation vs invariants: code contracts vs input validation.)

6. CRUD-y validations

Solution 4 (Execute / CanExecute pattern) works best in task-based scenarios where the user just wants to execute a task and where it's perfectly fine to return a generic error message if something goes wrong.

It's not as good in CRUD scenarios where you need to map potential validation errors back to UI. For example, you might have Delivery Time and Delivery Address UI fields which should be highlighted in case values in them are incorrect.

In such situation, you have two choices. The first one is to take the 2nd approach and do the validation in the application services layer. Again, not too pleasant but quite easy to implement.

The second one is to adjust the 4th solution and make the `CanDeliver` method return not just plain strings but special Error instances which would contain information about the source of the problem (read: UI fields). You may create an enum listing all those possible sources and then implement a mapping mechanism to convert them into field names on the UI.

Unlike in task-based validation, here, the choice is not as clear. The complexity overhead you'd need to introduce for the solution with `CanDeliver` method makes it less appealing for CRUD-y validations. Overall, both approaches are clunky, so you need to carefully weigh their pros and cons before considering one over another.

7. Summary

There are 4 possible ways to implement validation.

- `IsValid` method: transition an entity to a state (potentially invalid) and ask it to validate itself.
- Validation in application services.
- TryExecute pattern.
- Execute / CanExecute pattern.

There are two major types of validation.

- Task-based where you don't need to map validation errors to UI elements.
- CRUD-y where you do need to do that.

The Execute / TryExecute pattern works best for task-based scenarios. For CRUD scenarios, you need to choose between Execute / TryExecute and validation in application services. The choice comes down to purity versus ease of implementation.

8. Related articles

- Validation vs Invariants
- Validation in a DDD world

← Domain services vs Application services

Email uniqueness as an aggregate invariant →

Subscribe

I don't post everything on my blog. Don't miss smaller tips and updates. Sign up to my mailing list below.

Your email Sign up

Comments

63 Comments Enterprise Craftsmanship 1 Login

Recommend 10 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Marco Willemart • 5 years ago

This is indeed an interesting topic and one that is, in my opinion, not extensively covered in the various DDD books.

Personally I like to keep validation out of my domain model. Of course the domain model should enforce business rules and the like by means of invariants, pre and post conditions, etc. However validation is another concern that should be tackle elsewhere. I've found that adding validation to the domain model adds a lot of complexity into the code which quickly becomes hard to

Top