Design validations in the domain model layer

10/08/2018 • 5 minutes to read •  +1

In this article

In DDD, validation rules can be thought as invariants. The main responsibility of an aggregate is to enforce invariants across state changes for all the entities within that aggregate.

Domain entities should always be valid entities. There are a certain number of invariants for an object that should always be true. For example, an order item object always has to have a quantity that must be a positive integer, plus an article name and price. Therefore, invariants enforcement is the responsibility of the domain entities (especially of the aggregate root) and an entity object should not be able to exist without being valid. Invariant rules are simply expressed as contracts, and exceptions or notifications are raised when they are violated.

The reasoning behind this is that many bugs occur because objects are in a state they should never have been in.

Let's propose we now have a SendUserCreationEmailService that takes a UserProfile ... how can we rationalize in that service that Name is not null? Do we check it again? Or more likely ... you just don't bother to check and "hope for the best"—you hope that someone bothered to validate it before sending it to you. Of course, using TDD one of the first tests we should be writing is that if I send a customer with a null name that it should raise an error. But once we start writing these kinds of tests over and over again we realize ... "wait if we never allowed name to become null we wouldn't have all of these tests".

Implement validations in the domain model layer

Validations are usually implemented in domain entity constructors or in methods that can update the entity. There are multiple ways to implement validations, such as verifying data and raising exceptions if the validation fails. There are also more advanced patterns such as using the Specification pattern for validations, and the Notification pattern to return a collection of errors instead of returning an exception for each validation as it occurs.

Validate conditions and throw exceptions

The following code example shows the simplest approach to validation in a domain entity by raising an exception. In the references table at the end of this section you can see links to more advanced implementations based on the patterns we have discussed previously.

C# Copy

```
public void SetAddress(Address address)
{
    _shippingAddress = address?? throw new ArgumentNullException(nameof(address));
}
```

A better example would demonstrate the need to ensure that either the internal state did not change, or that all the mutations for a method occurred. For example, the following implementation would leave the object in an invalid state:

C# Copy

```
public void SetAddress(string line1, string line2,
    string city, string state, int zip)
{
    _shippingAddress.line1 = line1 ?? throw new ...
    _shippingAddress.line2 = line2;
    _shippingAddress.city = city ?? throw new ...
    _shippingAddress.state = (IsValid(state) ? state : throw new …);
}
```

If the value of the state is invalid, the first address line and the city have already been changed. That might make the address invalid.

A similar approach can be used in the entity's constructor, raising an exception to make sure that the entity is valid once it is created.

Use validation attributes in the model based on data annotations

Data annotations, like the Required or MaxLength attributes, can be used to configure EF Core database field properties, as explained in detail in the Table mapping section, but they no longer work for entity validation in EF Core (neither does the IValidatableObject.Validate method), as they have done since EF 4.x in .NET Framework.

Data annotations and the IValidatableObject interface can still be used for model validation during model binding, prior to the controller's actions invocation as usual, but that model is meant to be a ViewModel or DTO and that's an MVC or API concern not a domain model concern.

Having made the conceptual difference clear, you can still use data annotations and `IValidatableObject` in the entity class for validation, if your actions receive an entity class object parameter, which is not recommended. In that case, validation will occur upon model binding, just before invoking the action and you can check the controller's ModelState.IsValid property to check the result, but then again, it happens in the controller, not before persisting the entity object in the DbContext, as it had done since EF 4.x.

You can still implement custom validation in the entity class using data annotations and the `IValidatableObject.Validate` method, by overriding the DbContext's SaveChanges method.

You can see a sample implementation for validating `IValidatableObject` entities in this comment on GitHub. That sample doesn't do attribute-based validations, but they should be easy to implement using reflection in the same override.

However, from a DDD point of view, the domain model is best kept lean with the use of exceptions in your entity's behavior methods, or by implementing the Specification and Notification patterns to enforce validation rules.

It can make sense to use data annotations at the application layer in ViewModel classes (instead of domain entities) that will accept input, to allow for model validation within the UI layer. However, this should not be done at the exclusion of validation within the domain model.

Validate entities by implementing the Specification pattern and the Notification pattern

Finally, a more elaborate approach to implementing validations in the domain model is by implementing the Specification pattern in conjunction with the Notification pattern, as explained in some of the additional resources listed later.

It is worth mentioning that you can also use just one of those patterns—for example, validating manually with control statements, but using the Notification pattern to stack and return a list of validation errors.

Use deferred validation in the domain

There are various approaches to deal with deferred validations in the domain. In his book Implementing Domain-Driven Design , Vaughn Vernon discusses these in the section on validation.

Two-step validation

Also consider two-step validation. Use field-level validation on your command Data Transfer Objects (DTOs) and domain-level validation inside your entities. You can do this by returning a result object instead of exceptions in order to make it easier to deal with the validation errors.

Using field validation with data annotations, for example, you do not duplicate the validation definition. The execution, though, can be both server-side and client-side in the case of DTOs (commands and ViewModels, for instance).

Additional resources

- **Rachel Appel. Introduction to model validation in ASP.NET Core MVC**
  /aspnet/core/mvc/models/validation

- **Rick Anderson. Adding validation**
  /aspnet/core/tutorials/first-mvc-app/validation

- **Martin Fowler. Replacing Throwing Exceptions with Notification in Validations**
  https://martinfowler.com/articles/replaceThrowWithNotification.html

- **Specification and Notification Patterns**
  https://www.codeproject.com/Tips/790758/Specification-and-Notification-Patterns

- **Lev Gorodinski. Validation in Domain-Driven Design (DDD)**
  http://gorodinski.com/blog/2012/05/19/validation-in-domain-driven-design-ddd/

- **Colin Jack. Domain Model Validation**
  https://colinjack.blogspot.com/2008/03/domain-model-validation.html

- **Jimmy Bogard. Validation in a DDD world**
  https://lostechies.com/jimmybogard/2009/02/15/validation-in-a-ddd-world/

Previous Next

Is this page helpful?