DDD: What kinds of behavior should I put on a domain entity?

Asked 8 years, 5 months ago Active 6 years, 5 months ago Viewed 7k times

19

5

My team tries very hard to stick to Domain Driven Design as an architectural strategy. But, most of the time, our domain entities are pretty enimic. We'd like to be putting more business/domain behavior on our domain entities.

For example, Active Record puts data access on the entity. We don't want that because we happily use the repository pattern for data access.

Also, we design our software to be SOLID (the five software design principles that Uncle Bob put together). So, it's important to us that we pay attention to single responsibility, open-closed, liskov, interface segregation, and dependency inversion while designing our entities.

So, what kinds of behavior should we include? What kinds should we stay away from?

domain-driven-design entity solid-principles

asked Oct 16 '11 at 0:25
Byron Sommardahl
11.5k ● 13 ● 63 ● 124

Single responsibility is pretty antithetical to domain driven design. We had an interesting discussion about this at the NYCDDD meetup a couple months ago... – Domenic Oct 16 '11 at 0:46

1 I'd be interested in that discussion. I don't find that to be the case – Stephan Eggermont Aug 24 '12 at 18:22

27

It's been almost a year since I asked this question and I and my team have learned quite a lot since then. Here's how I would answer this question today:

The domain should represent (in code) what the business is or does (in real life). Domain entities, then, are the artifacts or actors found in that real-life business. What kind of behavior do those rea-life artifacts and actors have? All of it. In turn, what kind of behavior SHOULD domain entities have on them? All of it.

For instance, in real life, a manager can hire a new employee. The domain's representation of that should include entities like "manager" and "new employee". The manager is the actor, here.

```
//newEmployee comes from somewhere else... possibly the UI
//someManagerId comes from the logged in user
var manager = _repository.Get<Manager>(someManagerId);
manager.Hire(newEmployee);
```

So, the manager entity models/reflects the behavior of the real-life business, here. The alternative is to skip the manager entity as an actor, and push him off to the corner so a heavy-lifting "domain service" can do all the work... like this:

```
//newEmployeeService comes from somewhere else... possibly injected using IOC
newEmployeeService.Create(newEmployee, someManagerId);
```

In an anemic domain, you would use a domain service like this to create or hire an employee. It works, but it's not expressive and the behavior is not as discoverable. Who does what? Why is the manager required to create a new employee?

I think when I asked the question originally, I wanted to try to start including more behavior in my entities, but I really didn't know how without injecting services into my entities (for instance, with constructor injection). Since then, we've learned some new tricks and our team's entities are super-expressive. Here's, in a nutshell, what we are doing:

1. We try to, when possible, use actor entities to express the person or thing that is performing the action.
2. Actors have methods that express the actions they can perform
3. When a service is needed, it is injected as an argument in the method where it is used.
4. We fire domain events using BlingBag on every method on every domain entity to provide extensibility and give entities the ability to self-persist.

3 What about the newEmployee. Is it created by a factory or by the repository? Will it raise a domain event by creation? Should domain events be handled by repositories or should they go outside of the domain layer? I might ask too much :D – inf3rno Sep 26 '14 at 2:42

Im also trying to do the same. Like call = marketer.call(client); inquiry = marketer.makeInquiry(client, date); im curious how did you configure your Manager in ORM and the design of the database. Is manager in a Single Table Inheritance? I am doing single table inheritance but I dont need the discriminator column. basically the actors have the same data with different roles only. – Jaime Sangcap Feb 19 '15 at 9:05

1 +1 for your point #3 (When a service is needed, it is injected as an argument in the method where it is used). I've seen this subtlety elude a lot of people (including myself). When you think about it, it makes sense to organise it this way rather than using constructor injection as it makes it clear what the dependencies of each action are. – MetaFight Dec 12 '17 at 18:43

4

If you have to ask what behaviour you should put on the domain entity then you probably don't need DDD. I'm trying to be helpful here, because I have had a lot of pain trying to fit DDD into a place it didn't belong.

DDD or even domain model are patterns that can be followed *after* it is discovered that the domain complexity is too high for any other pattern to work. So just CRUD is not suitable for DDD. From my understanding, DDD fits when you have a bounded context that contains *complex* business rules that need to be run before transitioning state for the aggregate root. So I would not include validation in the definition of complex.

The kind of behaviour that you want to put in your entities is intimately related to the business problem you are attempting to solve. The concern about persistence (repository etc) should come after (In fact, persistence might be in an workflow or event store).

Hope this helps.

answered Mar 8 '12 at 14:08
Davin Tryon
58.4k ● 10 ● 131 ● 121

1

some behavior that i try to put into my domain, entities or value objects.

validation before persistence. validation before transition into a new state. For example order aggregate root entity can validate its internal state and its aggregate children before going into state Sent. minimize get set properties and use value objects as much as you can. First it makes the model richer with behavior. the entities get more descriptive. second you more rarely put your entity into a

edited Dec 23 '11 at 11:17
Be.St.
3,773 ● 2 ● 21 ● 35

answered Oct 17 '11 at 18:36
Magnus Backeus
1,848 ● 12 ● 21

email me if would like to exchange more deeply discussions about ddd and behavior. – Magnus Backeus Oct 17 '11 at 18:37

0

The behaviour that is on your entities should reflect the business model. What can be done to or by that entity is the business world should be sopmething that can be done to or by the entity class. For example:

In an online shopping system, you can add a product to your cart. So the Cart class should look like this:

```
public class Cart
{
    //...

    public void AddProduct(Product product)
    {
        ...code to add product to cart.
    }
}
```

It could be argued that the methids should reflect the use cases.

answered Mar 8 '12 at 13:20
Paul T Davies
2,227 ● 1 ● 15 ● 31

But if product represents another aggregate root, you should only reference by Id, not the instance. Additionally Product being an aggregate root doesn't seem to fit `Product.AddToCart` that modelling scenario – Andez Jan 25 '19 at 13:23

@Andez why do you say you should reference by id and not by reference? In DDD the opposite is true. – Paul T Davies Jan 26 '19 at 14:40

In Evans blue book and Vernon's red book (p359), the guidance is not to refer to an aggregate by reference, you refer to it by Id. To quote: `When designing Aggregates, we may desire a deep traversal through deep object graphs, but that is not the motivation of the pattern.`

This sounds like Vernon's advice not Evans'. I'm not sure what problem it solves. I often reference one aggregate from another and it has never caused me any problems. If business logic affects two aggregates then it is necessary. Some advice seems to suggest you can avoid it by having services that coordinate the two aggregates, but I try to avoid too many services as some devs can start putting everything in services, leading to an anaemic domain models. – Paul T Davies Jan 27 '19 at 21:46