SOFTWARE ENGINEERING

Domain Services vs. Factories vs. Aggregate Roots

Asked 7 years ago Active 5 years, 4 months ago Viewed 8k times

8

4

After dealing with DDD for months now, I'm still confused about the general purposes of domain services, factories and aggregate roots in relation to each other, e.g. where they overlap in their responsibility.

Example: I need to 1) create a complex domain entity in a saga (process manager) which is followed by 2) a certain domain event that needs to be handled elsewhere whereas 3) the entity is clearly an aggregate root that marks a bounded context to some other entities.

1. The factory IS be responsible for creation of the entity/aggregate root
2. The service CAN create the entity, since he also throws the domain event
3. The service CAN act as a aggregate root (create 'subentity' of 'entity' with ID 4)
4. The aggregate root can create and manage 'subentities'

When I introduce the concept of a aggregate root as well as a factory to my domain, a service seems no longer needed. However, if I'm not, the service can handle everything needed as well with the knowledge and dependencies he has.

Code Example *based on the ubiquitous language of a car repair shop*

Code Example *based on the ubiquitous language of a car repair shop*

```
public class Car : AggregateRoot {

    private readonly IWheelRepository _wheels;
    private readonly IMessageBus _messageBus;

    public void AddWheel(Wheel wheel) {
        _wheels.Add(wheel);
        _messageBus.Raise(new WheelAddedEvent());
    }

}

public static class CarFactory {

    public static Car CreateCar(string model, int amountofWheels);


}
```

..or...

```
public class Car {

    public ICollection<Wheel> Wheels { get; set; }

}

public interface ICarService {

    Car CreateCar(args);
    void DeleteCar(args);
    Car AddWheel(int carId, Wheel wheel);

}
```

design-patterns domain-driven-design separation-of-concerns cqrs

Share Improve this question Follow

edited May 5 '14 at 7:51

asked May 5 '14 at 7:44

Acrotygma
301 2 7

I use domain services to implement domain use cases. I rarely use factories, don't really need them. – MikeSW May 6 '14 at 13:41

2 Answers

Active Oldest Votes

8

> After dealing with DDD for months now, I'm still confused about the general purposes of domain services, factories and aggregate roots in relation to each other, e.g. where they overlap in their responsibility.

The aggregate root is responsible for ensuring that the state is consistent with the business invariant. In CQRS lingo, you change the model by issuing commands to an aggregate root.

The domain service is a query support mechanism for the aggregates. For example, a domain service might support a calculation. The command handler passes the domain service to the aggregate, the aggregate (processing a command) needs the result of the calculation, so it will pass to the domain service the parts of its state that are needed as inputs to the calculation. The domain service makes the calculation and returns the result, and the aggregate then gets to decide what to do with the result.

"Factory" can mean a couple of things. If you are using the factory to create an entity that is within the aggregate boundary, then it's just an implementation detail -- you might use it if the construction of the object is complicated.

In some circumstances, the term "Repository" is used. This usually implies that it is part of the persistence layer (not part of the domain model) responsible for creating aggregate roots. Roughly, the command handler (which is part of the application) validates a command, then uses the Repository to load the aggregate root that the command is addressed to. The command handler will then invoke the specified method on the aggregate root, passing in parameters from the command object, and possibly passing in the domain service as an argument as well.

In your examples, the key question to ask is where the responsibility for deciding whether a command should be run lives. The application is responsible for making sure that the command is well formed (all the command data is present, the data was translated into values recognized by the domain without throwing validation errors, and so on). But who gets to decide "No, you don't get to add a wheel right now -- the business rules don't allow it!"

In the DDD world, that is unquestionably the responsibility of the aggregate root. So any logic to determine that should be in the aggregate root, and the ICarService goes away.

(The alternative implementation, where the aggregate exposes its state, and the cars service checks the business rules and manipulates the state of the object, is regarded as an anti pattern -- an example of an "anemic" aggregate. "Setters" in an aggregate is a code smell. "Getters" in an aggregate are often a code smell -- especially in CQRS, where the responsibility for supporting queries is supposed to be "somewhere else" -- in the read model.)

Share Improve this answer Follow

answered Jan 12 '16 at 1:23
VoiceOfUnreason
26.7k 1 34 62

Isn't it just a state of mind ? Let's say for an entity you have an entityService, an entityRepository and the entity itself. You could say the "aggregate" is the combination of the entityService + entity, there is no "exposure" of the entity to other micro-services which will see only the entityService API, nothing else. – Tristan Jun 7 '20 at 12:10

Not really, because "aggregate" means something; to be precise, it is a label we use when referring to the life cycle management pattern described in chapter 6 of the Evans book. – VoiceOfUnreason Jun 7 '20 at 12:17

Yes I've read the book 10 years ago, but since then I haven't seen any implementation of "aggregate" and I haven't seen any issue about it, so I concluded what is important is to have clean structuration of code and to keep it related to the functional specifications. "Repositories" in the contrary have become very common thanks to the implementation in Spring Data. – Tristan Jun 7 '20 at 12:23

7

DDD is in part a reaction to anaemic domain models, where your entities would only have state, but not behaviour.

It's true that in a sense you could put all the behaviour of Car in a separate service, but why would you want to? For that to work, you'd need to expose all sorts of state in Car, which you would normally like to keep private (like Wheels).
If you expose Wheels like that, any code could do all sorts of funky stuff to that collection, outside of any normal business flow. Keep in mind that the point of an aggregate is to have something that is transactionally consistant between business flow transactions. Exposing Wheels like that completely undermines that safety.
For example: say your business only wants to support cars with four wheels. If you encapsulate access to Wheels, you can enforce that. If you don't, it's completely possible to add a hundred wheels to a car, because Wheels would be exposed.

A service is generally used as a coordinator. A commandhandler (assuming something like CQRS) translates a command object to more strongly typed parameters, reducing primitive obsession. Then it can call a service with those more "domain-ish" parameters.

The service retrieves entities from repositories and invokes behaviours on them. Depending on your architecture, it can then collect any changes (events) and pass them to a bus or whatever.
In more advanced scenarios, you'd use a unit of work that keeps track of all your entities and can collect the changes from all of them in one swoop.

As for a factory (as an aside), nothing's stopping you from adding a static factory method your aggregate class, instead of creating a separate factory.

Share Improve this answer Follow

edited May 5 '14 at 8:25

answered May 5 '14 at 8:15

Stefan Billiet
3,409 1 15 14

"DDD is in part a reaction to anaemic domain models, where your entities would only have state, but not behaviour." Your explaination sounds reasonable. I see more and more that DDD is about thinking in DDD, not doing DDD. The only concern I have with your suggestion is when I want to expose parts of my domain to other applications (let's say I've got a frontend application and a backend service which communicate over WCF), then I need to introduce new models just for the sake of communication (because the frontend might not know what a aggregate is and what behavior it has) – Acrotygma May 5 '14 at 9:09

@Acrotygma Odds are that your frontend will need data in a completely different form than the form in which your aggregates will be modelled. This is almost always the case, so you're better off making that distinction explicit and creating separate write and read models. I know this sounds like a lot of work, but you can't have one model that suits all scenarios; in the best case, you'll wind up with something not particularly suited for read nor write. E.g. aggregates are great for business logic, not so much for sending accross the wire as a DTO. – Stefan Billiet May 5 '14 at 9:19