SUBSCRIBE

Services: Infrastructure, Application, Domain, and Object

DEC 30, 2008

You may have read on various blog from individuals--including well-known master craftsmen--that having references to a service or repository inside of your domain entity is an anti-pattern or a code smell which leads to the dreaded "anemic domain model" problem. If you agreed with this, you'd be wrong.

Allow me qualify the above statement. The issue isn't so much having services inside your domain entities--or value objects for that matter--it's having the wrong kind of services. Before we elaborate on the various types of services we need to understand what a service actually is.

What is a Service?

A service, in the simplest sense, is an object that does something. One of the single-most identifying features of a service is that it is stateless. In other words it doesn't inherently track state internally even though it may make use of external factors to determine proper outcome. A service call is thus non-deterministic--potentially having a different result for each call--but the determinism is external to the service.

There are several kinds of services: infrastructure services, application services, domain services, and little-known "object" services--each having a different purpose and reason for existence.

Infrastructure Services

This is probably the most easily identifiable because it deals with infrastructure concerns such as the interacting with a database, communicating across a network, sending an email, etc. If not carefully isolated, infrastructure concerns can bleed into the domain model and make testability virtually impossible and erecting barriers to change.

Keep this kind of service out of entity objects.

Application Services

Another common and well-known service is that of the application service. This one is often blamed for the anemic domain model--and rightly so. When this service overused or given too much responsibility it robs the domain model of its power and and strength.

This is not to say that application services are bad. In fact, application services are often the best way to orchestrate a complex unit of work amongst the various domain objects.

Keep this kind of service out of entity objects.

Domain Services

This is the oft-forgotten, oft-overloaded variety of services. In Evans' landmark book *Domain-Driven Design* he dedicates an entire section to domain services validating that services are, indeed, first-class citizens in the domain. The biggest distinguishing feature of domain service is that that it is inherently part of the "ubiquitous language".

Domain services may make use of whatever object necessary, including other domain services, repositories, entities--even infrastructure services--to accomplish their assigned responsibility. Furthermore because they are **inside** of the domain they may be freely called **from** other domain objects--including entity and value objects, among others.

Another Kind of Service?

This is a term I have coined to help describe a particular kind of services that, likes its siblings, is stateless and has a specific responsibility. This service is often confused with an application service or even an infrastructure service. Perhaps a small, contrived example will suffice:

Suppose an entity has the responsibility to perform a complex calculation. Naturally this functionality rightfully belongs to the entity. Now suppose in our discovery of the ubiquitous language that another entity has need to perform the same calculation.

Do we simply copy and paste the calculation code into the other entity because "services aren't allowed in entities"? Do we make the complex calculation method as *public static* on one entity and call it from the new entity? The logical answer is neither.

Pragmatically speaking, we want to conform to one of the most well-understood principles in software development: Don't Repeat Yourself.

Copying and pasting code from class to class stands in direct violation of this principle and creates a stench bordering on unforgivable.

Object Services

Because of this there is a need for a fourth kind of service: the object service. The idea behind this service is that it performs the function that was originally a part of a single domain entity, but to avoid duplication and to promote composability and reuse, it has been factored out to another object.

In its simplest form an object service would receive primitives into its method call and return some kind of primitive. In a more complex scenario it would define an interface as a parameter and accept the entity which contains it as a parameter in order to prevent coupling.

To some this may seem as though we have weakened our domain model to the point where it becomes anemic, being little more than DTOs. The fact that the logic is not coded in the same class file as the domain entity does not mean the model is anemic. A DTO is a series of getters and setters (in C#) or a bunch of member variables with no ability to act but only to be acted upon.

A domain entity has the ability to act, but more especially to **delegate and coordinate** subordinate logic to other objects under its control. In this way it is externally impossible to distinguish whether the domain entity is performing all of the logic itself or delegating the work to other objects--much the same way we think of a car as a single, mental unit without giving much thought to all of the subordinate parts that make it useful.

Perhaps some of the challenge and reasoning behind keeping all services out of domain entities is because of the challenge of properly injecting services into an entity when using an OR/M. This is a purely technical consideration and is more a manifestation of the limits of our current technology. As such this shouldn't even be a consideration.

Conclusion

Delegation, a foundational pillar of *GoF Patterns*, never made a domain model anemic or created a DTO from a proper domain entity. Object services, when properly identified and isolated, represent an easy and powerful way to ensure a strong domain model by ensuring testability, separation of concerns, delegation, and code de-duplication, which are explicitly or inherently SOLID principles.

Jonathan Oliver

Share this post

Read more posts by this author.

Utah, USA *https://blog.jonathanoliver.com/*

4 Comments **Jonathan Oliver's Blog** **Disqus' Privacy Policy** 1 **Login**

Recommend 1 Tweet Share **Sort by Oldest**

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

Nick • 11 years ago

Great post Jonathan. Is an Object Service something along the lines of a Strategy by this definition? Would be interesting to see an example.

Cheers,
Nick

Reply • Share ›

Jonathan Oliver • 11 years ago

@Nick,

The Strategy Pattern is typically used to vary a particular implementation at runtime, very much like dependency injection. The idea behind my post is that we shouldn't be afraid to have service objects inside of what Eric Evans calls a domain entity.

Regarding the example, I had a few but they're rather contrived. Without the surrounding background information they can easily be pulled apart.

Reply • Share ›

Nick • 11 years ago

Yeah, that's the painful thing about examples and abstract concepts :) I totally agree with your point - just checked out Fabio's post too, great to see those changes coming through in NH!

Reply • Share ›

Anonymous • 10 years ago

Jonathan,

First of all I don't agree that people whining about domain objects depending on services use Anemic Domain Model as an argument. On the contrary, they are not aware of this problem problem, since entities without dependencies (which they are postulating) end up being mere DTOs.

I'd really like to avoid ADM, so, to be sure - are you saying that Entity object itself should be able to interact with database (via domain model service which in turn uses repository)?

I almost hear all the people saying that it breaks Separation of Concerns, but I think you're right: merely using something, that interacts with db internally doesn't mean that it's Entity's concern to interact with it.

• Reply • Share ›

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Do Not Sell My Data**

READ THIS NEXT

YOU MIGHT ENJOY

Windows Server 2008 x64 & CERC 1.5 6-Channel SATA RAID

Assembly & Artifact Storage