# Services in Domain-Driven Design

21 August, 2008. It was a Thursday.

Services are first-class citizens of the domain model.  When concepts of the model would distort any Entity or Value Object, a Service is appropriate.  From Evans' DDD, a good Service has these characteristics:

- The operation relates to a domain concept that is not a natural part of an Entity or Value Object
- The interface is defined in terms of other elements in the domain model
- The operation is stateless

Services are always exposed as an interface, not for "swappability", testability or the like, but to expose a set of cohesive operations in the form of a contract.  On a sidenote, it always bothered me when people say that an interface with one implementation is a design smell.  No, an interface is used to expose a contract.  Interfaces communicate design intent, far better than a class might.

But most examples I see of Services are something trivial, such as IEmailSender.  But Services exist in most layers of the DDD layered architecture:

- Application
- Domain
- Infrastructure

An Infrastructure Service would be something like our IEmailSender, that communicates directly with external resources, such as the file system, registry, SMTP, database, etc.  Something like NHibernate would show up in the Infrastructure.

Domain services are the coordinators, allowing higher level functionality between many different smaller parts.  These would include things like OrderProcessor, ProductFinder, FundsTransferService, and so on.  Since Domain Services are first-class citizens of our domain model, their names and usages should be part of the Ubiquitous Language.  Meanings and responsibilities should make sense to the stakeholders or domain experts.

In many cases, the software we write is replacing or supplementing a human's job, such as Order Processor, so it's often we find inspiration in the existing business process for names and responsibilities.  Where an existing name doesn't fit, we dive into the domain to try and surface a hidden concept with the domain expert, which might have existed but didn't have a name.

Finally, we have Application Services.  In many cases, Application Services are the interface used by the outside world, where the outside world can't communicate via our Entity objects, but may have other representations of them. Application Services could map outside messages to internal operations and processes, communicating with services in the Domain and Infrastructure layers to provide cohesive operations for outside clients.  Messaging patterns tend to rule Application Services, as the other service layers don't have a reference back out to the Application Services.  Business rules are not allowed in an Application Service, those belong in the Domain layer.

In top-down design, we typically start from the Application or Domain Service, defining the actual interface clients use, then use TDD to drive out the implementation.  As we're always starting from the client perspective with actual client scenarios, we get a high degree of confidence that what we're building will create success and add value.  When stories are vertical slices of functionality, this is fairly straightforward, at least mechanically so.

I used to make the mistake of dismissing Services as a necessary evil, confined to the Infrastructure layer.  Through more reading and conversation through our recent Austin DDD Book Club, I've started to realize the potential the Application and Domain services have in creating a well-designed model.

← DDD, Repositories and ORMs Parameter lists in NHibernate →

**26 Comments** **Jimmy Bogard Blog** **Disqus' Privacy Policy** 1 **Login**

Recommend 5 Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

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

## Recent Author Posts

- Immutability in DTOs?
- MediatR 8.0 Released
- User Secrets in Docker-based .NET Core Worker Applications
- Contoso University Vertical Slice App Updated to ASP.NET Core 3.0
- Document-Level Optimistic Concurrency in MongoDB
- Building Messaging Endpoints in Azure: Functions
- Building Messaging Endpoints in Azure: Container Instances
- Building Messaging Endpoints in Azure: WebJobs
- Integration Testing with xUnit
- AutoMapper LINQ Support Deep Dive

## Recent Site Posts

- Immutability in DTOs?
- MediatR 8.0 Released

- User Secrets in Docker-based .NET Core Worker Applications
- Contoso University Vertical Slice App Updated to ASP.NET Core 3.0
- Document-Level Optimistic Concurrency in MongoDB
- Building Messaging Endpoints in Azure: Functions
- Building Messaging Endpoints in Azure: Container Instances
- Building Messaging Endpoints in Azure: WebJobs
- Integration Testing with xUnit
- AutoMapper LINQ Support Deep Dive

## Authors

- Andrew Siemer
- Chad Myers
- Chris Missal
- Chris Patterson
- Derek Greer
- Derik Whittaker
- Eric Anderson
- Eric Hexter
- Gabriel Schenker
- Gregory Long
- Hugo Bonacci
- James Gregory
- Jason Meridth
- Jimmy Bogard
- John Teague
- Josh Arnold
- Joshua Flanagan
- Joshua Lockwood
- Keith Dahlby

- Matt Hinze
- Patrick Lioi
- Rod Paddock
- Ryan Rauh
- Ryan Svihla
- Scott Densmore
- Sean Biefeld
- Sean Chambers
- Sharon Cichelli
- Steve Donie