Domain Driven Design: Domain Service, Application Service

Asked 10 years ago Active 5 days ago Viewed 63k times

248

Can someone explain the difference between domain and application services by providing some examples? And, if a service is a domain service, would I put the actual implementation of this service within the domain assembly and if so, would I also inject repositories into that domain service? Some info would be really helpful.

architecture domain-driven-design

163

edited Oct 19 '14 at 23:10

Harshdeep
4,474 8 27 41

asked Feb 15 '10 at 20:29

Chris
5,721 4 35 57

8 Answers

337

Services come in 3 flavours: **Domain Services**, **Application Services**, and **Infrastructure Services**.

- **Domain Services** : Encapsulates *business logic* that doesn't naturally fit within a domain object, and are **NOT** typical CRUD operations – those would belong to a *Repository*.
- **Application Services** : Used by external consumers to talk to your system (think *Web Services*). If consumers need access to CRUD operations, they would be exposed here.
- **Infrastructure Services** : Used to abstract technical concerns (e.g. MSMQ, email provider, etc).

Keeping Domain Services along with your Domain Objects is sensible – they are all focused on domain logic. And yes, you can inject Repositories into your Services.

Application Services will typically use both Domain Services *and* Repositories to deal with external requests.

Hope that helps!

edited Jan 15 '19 at 17:04

Darryl Hein
127k 85 199 253

answered Feb 17 '10 at 10:07

Vijay Patel
13.4k 6 26 32

2 Where would you put the commands and queries by CQRS? Which service generates them and which service handles them? – inf3rno Sep 20 '14 at 23:42

5 I think Application Services should be indepent of technical details like "web services", they are used by such services. See Services in Domain-Driven Design – deamon Apr 13 '15 at 15:17

1 @prograhammer - An example for a domain service could be FundsTransferService, where the domain model is a BankAccount, the transfer could have some business logic that doesn't fit directly into an account object (taken from Evans DDD book). – BornToCode Mar 29 '16 at 18:11

so say for example Loginuser() would be an example of an domain service. where as getUsers() would be an application service?? – filthy_wizard Oct 11 '16 at 11:07

Both are rather application services because authentication and often also authorization decisions do not belong to the core domain. – MauganRa Aug 6 '17 at 18:44

(If you don't feel like reading, there's a summary at the bottom :-)

110

I too have struggled with the precise definition of application services. Although Vijay's answer was very helpful to my thinking process a month ago, I have come to disagree with part of it.

Other resources

There's very little information about application services. Subjects like aggregate roots, repositories and domain services are discussed extensively, but application services are only mentioned briefly or left out altogether.

The MSDN Magazine article An Introduction To Domain-Driven Design describes application services as a way to transform and/or expose your domain model to external clients, e.g. as a WCF service. This is how Vijay describes application services too. From this

point of view, application services are an **interface to your domain**.

Jeffrey Palermo's articles on the Onion Architecture (part one, two and three) are a good read. He treats application services as **application-level concepts**, such as a user's session. Although this is closer to my understanding of application services, it's still not in line with my thoughts on the subject.

My thoughts

I have come to think of application services as **dependencies provided by the application**. In this case the application could be a desktop application or a WCF service.

Domain

Time for an example. You start out with your domain. All entities and any domain services that don't depend on external resources are implemented here. Any domain concepts that depend on external resources are defined by an interface. Here is a possible solution layout (project name in bold):

```
My Solution
- My.Product.Core (My.Product.dll)
  - DomainServices
      IExchangeRateService
    Product
    ProductFactory
    IProductRepository
```

The `Product` and `ProductFactory` classes have been implemented in the core assembly. The `IProductRepository` is something that is probably backed by a database. The implementation of this is not the domain's concern and is therefore defined by an interface.

For now, we'll focus on the `IExchangeRateService`. The business logic for this service is implemented by an external web service. However, its concept is still part of the domain and is represented by this interface.

Infrastructure

The implementation of the external dependencies are part of the application's infrastructure:

```
My Solution
+ My.Product.Core (My.Product.dll)
- My.Product.Infrastructure (My.Product.Infrastructure.dll)
  - DomainServices
      XEExchangeRateService
    SqlServerProductRepository
```

`XEExchangeRateService` implements the `IExchangeRateService` domain service by communicating with xe.com. This implementation can be used by your applications that utilize your domain model, by including the infrastructure assembly.

Application

Note that I haven't mentioned application services yet. We'll look at those now. Let's say we want to provide an `IExchangeRateService` implementation that uses a cache for speedy lookups. The outline of this decorator class could look like this.

```
public class CachingExchangeRateService : IExchangeRateService
{
    private IExchangeRateService service;
    private ICache cache;

    public CachingExchangeRateService(IExchangeRateService service, ICache cache)
    {
        this.service = service;
        this.cache = cache;
    }

    // Implementation that utilizes the provided service and cache.
}
```

Notice the `ICache` parameter? This concept is not part of our domain, so it's not a domain service. It's an **application service**. It's a dependency of our infrastructure that may be provided by the application. Let's introduce an application that demonstrates this:

```
My Solution
- My.Product.Core (My.Product.dll)
  - DomainServices
      IExchangeRateService
    Product
    ProductFactory
    IProductRepository
- My.Product.Infrastructure (My.Product.Infrastructure.dll)
  - ApplicationServices
      ICache
  - DomainServices
      CachingExchangeRateService
      XEExchangeRateService
    SqlServerProductRepository
- My.Product.WcfService (My.Product.WcfService.dll)
  - ApplicationServices
      MemcachedCache
    IMyWcfService.cs
```

```
+ MyWcfService.svc
+ Web.config
```

This all comes together in the application like this:

```
// Set up all the dependencies and register them in the IoC container.
var service = new XEExchangeRateService();
var cache = new MemcachedCache();
var cachingService = new CachingExchangeRateService(service, cache);

ServiceLocator.For<IExchangeRateService>().Use(cachingService);
```

Summary

A complete application consists of three major layers:

- domain
- infrastructure
- application

The domain layer contains the domain entities and stand-alone domain services. Any domain *concepts* (this includes domain services, but also repositories) that depend on external resources, are defined by interfaces.

The infrastructure layer contains the implementation of the interfaces from the domain layer. These implementations may introduce new *non-domain* dependencies that have to be provided the application. These are the application services and are represented by interfaces.

The application layer contains the implementation of the application services. The application layer may also contain additional implementations of domain interfaces, if the implementations provided by the infrastructure layer are not sufficient.

Although this perspective may not match with the general DDD definition of services, it does separate the domain from the application and allows you to share the domain (and infrastructure) assembly between several applications.

answered Mar 9 '11 at 21:53

Niels van der Rest
26.1k 14 72 84

2 @dario-g: You'd have to reconstruct/repopulate your domain model from the request model and pass the domain model to the domain service. This question may provide you with some ideas. If not, let me know and I'll see if I have some time to add an answer to the other question. – Niels van der Rest Mar 9 '11 at 23:00

1 @Tiendq: Do you mean the `IExchangeRateService` interface? This is a domain concept, i.e. something that is included in your customer's ubiquitous language. Other parts of your domain may depend on this service, which is why its interface in defined in the domain layer. But because its implementation involves an external web service, the implementing class resides in the infrastructure layer. This way the domain layer is only concerned with business logic. – Niels van der Rest Mar 12 '11 at 13:20

4 @Tiendq: In a traditional layered architecture, the infrastructure is usually domain-agnostic. But in the Onion Architecture (see links in my answer) the infrastructure implements the domain's external dependencies. But I wouldn't say the infrastructure *depends* on the domain, it just *references* it. I have taken the term 'infrastructure' from the Onion Architecture, but 'externals' may be a better name. – Niels van der Rest Mar 15 '11 at 8:21

1 @Derek: One of those 'things' could be an `ExchangeRate` instance, which contains a base currency, a counter currency and and the exchange rate value between these two currencies. These tightly related values represent the 'exchange rate' concept from the domain, so these live in the domain layer. Although it may seem like a simple DTO, in DDD it is called a Value Object and it could contain additional business logic for comparing or transforming instances. – Niels van der Rest Jan 23 '12 at 9:26

6 I disagree with the part where you disagree with Vijay and here's why. CachingExchangeRateService is an infrastructure concern. Even though you're generically accepting an ICache, the implementation for that ICache depends on the technology involved (ie. Web, Windows). Just because it's generic does not make it an application service. An application service is your domain's API. What if you wanted to reveal your domain to someone else writing an app, what will they use? Application Services, and they may not need caching so your caching impl is useless to them (ie.why it's infrastructure) – Aaron Hawkins Dec 27 '12 at 21:57

38

The best resource that helped me understand the difference between an Application Service and a Domain Service was the java implementation of Eric Evans' cargo example, found here. If you donwload it, you can check out the internals of RoutingService (a Domain Service) and the BookingService, CargoInspectionService (which are Application Services).

My 'aha' moment was triggered by two things:

- Reading the description of the Services in the link above, more precisely this sentence:

  > Domain services are expressed in terms of the ubiquitous language and the domain types, i.e. the method arguments and the return values are proper domain classes.

- Reading this blog post, especially this part:

  > What I find a big help in separating the apples from the oranges is thinking in terms of application workflow. All logic concerning the application workflow typically end up being Application Services factored into the Application Layer, whereas concepts from the domain that don't seem to fit as model objects end up forming one or more Domain Services.

edited Feb 26 at 18:12 Benjamin W.

answered Aug 12 '11 at 10:50 Ghola

26.4k 15 62 67

508 1 6 13

2 I agree, this is exactly how I define Application Services, and it fit all the situations I have met so far. Domain Services deal with everything related to domain objects, but that go beyond the scope of a single entity. Ex: BookReferencesService.GetNextAvailableUniqueTrackingNumber(), the focus is clearly business rules*. Regarding Application Service, it's exactly what you describe, most of the time I begin by putting this business workflow into my controller actions, and when I notice it I refactor this logic in the application service layer. We might say that this layer is for use cases – tobiak777 Feb 18 '15 at 14:02

1 *And such domain service interfaces are consumed by the domain entities. – tobiak777 Feb 18 '15 at 14:05

30

From the Red Book (Implementing Domain Driven Design, by Vaughn Vernon), this is how I understand the concepts:

Domain objects (**entities** and **value objects**) encapsulate behavior required by the (sub)domain, making it natural, expressive, and understandable.

Domain services encapsulate such behaviors that do not fit in a *single* domain object. For example, a book library lending a `Book` to a `Client` (with corresponding `Inventory` changes) might do so from a domain service.

Application services handle the flow of use cases, including any additional concerns needed *on top of* the domain's. It often exposes such methods through its API, for consumption by external clients. To build on our previous example, our application service might expose a method `LendBookToClient(Guid bookGuid, Guid clientGuid)` that:

- Retrieves the `Client`.
- Confirms its permissions. (*Note how we have kept our domain model free of security / user management concerns. Such pollution could lead to many problems. Instead, we fulfill this technical requirement here, in our application service.*)
- Retrieves the `Book`.
- Calls the domain service (passing the `Client` and `Book`) to handle the *actual domain logic* of lending the book to the client. For instance, I imagine that confirming the book's availability is definitely part of the domain logic.

An application service should generally have a very simple flow. Complex application service flows often indicate that domain logic has leaked out of the domain.

As you can hopefully see, the **domain model** stays very *clean* this way, and is easy to understand and discuss with the domain experts, because it only contains its own, actual business concerns. The **application flow**, on the other hand, is *also* much easier to manage, since it is relieved of domain concerns, and becomes concise, and straightforward.

edited Mar 19 '18 at 16:21 answered Jan 3 '17 at 13:42

Timo
4,691 3 29 41

2 I would say that the **application service** is also the point where dependencies are resolved. Its method is a use case, a single flow, so it can make informed decisions on concrete implementations to use. Database transactions fit here as well. – Timo Jan 3 '17 at 13:54

28

Domain service is the extension of the domain. It should be seen only in the context of the domain. This is not some user action like for instance *close account* or something. The domain service fits where there is no state. Otherwise it would be a domain object. Domain service does something which makes sense only when being done with other collaborators (domain objects or other services). And that *making sense* is the responsibility of another layer.

Application service is that layer which initializes and oversees interaction between the domain objects and services. The flow is generally like this: get domain object (or objects) from repository, execute an action and put it (them) back there (or not). It can do more - for instance it can check whether a domain object exists or not and throw exceptions accordingly. So it lets the user interact with the application (and this is probably where its name originates from) - by manipulating domain objects and services. Application services should generally represent all possible *use cases*. Probably the best thing you can do before thinking about the domain is to create application service interfaces what will give you a much better insight in what you're really trying to do. Having such knowledge enables you to focus on the domain.

Repositories can generally speaking be injected into domain services but this is rather rare scenario. It is the application layer who does it most of the time though.

answered Aug 2 '14 at 12:32

kboom
1,545 3 20 34

10 "The domain service fits where there is no state. Otherwise it would be a domain object." made it click for me. Thank you. – Nick Nov 10 '15 at 18:03

9

Domain Services : Methods that don't really fit on a single entity or require access to the repository are contained within domain services. The domain service layer can also contain domain logic of its own and is as much part of the domain model as entities and value objects.

Application Services : The Application service is a thin layer that sits above the domain model and coordinates the application activity. It does not contain business logic and does not hold the state of any entities; however, it can store the state of a business

> workflow transaction. You use an Application service to provide an API into the domain model using the Request-Reply messaging pattern.

Millett,C (2010). Professional ASP.NET Design Patterns. Wiley Publishing. 92.

answered Jan 17 '17 at 10:35

GorkemHalulu
2,195 1 20 23

5

Domain Services: A service that expresses a *business logic* that is not part of any Aggregate Root.

- You have 2 Aggregate:
  - `Product` which contains name and price.
  - `Purchase` which contains purchase date, list of products ordered with quantity and product price at that time, and payment method.
- `Checkout` is not part of either of these two models and is concept in your business.
- `Checkout` can be created as a Domain Service which fetches all product and compute the total price, pay the total by calling another Domain Service `PaymentService` with an implementation part of Infrastructure, and convert it into `Purchase`.

Application Services: A service that *"orchestrates"* or exercises Domain methods. This can be as simple as just your Controller.

This is the place where you usually do:

```
public String createProduct(...some attributes) {
  if (productRepo.getByName(name) != null) {
    throw new Exception();
  }

  productId = productRepository.nextIdentity();

  product = new Product(productId, ...some attributes);

  productRepository.save(product);

  return productId.value();
  // or Product itself
  // or just void if you dont care about result
}
```

```
public void renameProduct(productId, newName) {
  product = productRepo.getById(productId);

  product.rename(newName);

  productRepo.save(product);
}
```

You can do validations here like checking if a `Product` is unique. Unless a `Product` being unique is an invariant then that should be part of Domain Service that might be called `UniqueProductChecker` because it can't be part of `Product` class and it interacts with multiple Aggregates.

Here is full-blown example of DDD project: https://github.com/VaughnVernon/IDDD_Samples

You can find lots of examples of Application Service and a couple of Domain Service

edited Jun 1 '19 at 18:45

answered Jun 1 '19 at 18:39

doesnotmatter
171 2 7

Is it mandatory to validate and save entities only in Application Services? If I have entities A, B and C and all of them related to each other (A -> B -> C) and operation on A should cause changes to B and C by calling one Domain Service from another, how to do it? – MrNVK Jul 17 '19 at 12:24

> Is it mandatory to validate and save entities only in Application Services? If you have to, then yes. Most of the times you have to check if an ID exists because otherwise you'll work on a null variable. – doesnotmatter Jul 17 '19 at 13:39

1 > If I have entities A, B and C and all of them related to each other (A -> B -> C) and operation on A should cause changes to B and C by calling one Domain Service from another, how to do it? I'm not sure what you mean by "calling one Domain Service from another", but for reactions to changes of an Entity, you can use Events or just orchestrate it with Application service like: aggregateA.doOperation(), aggregateB.doAnother(). Search for: Orchestration vs Choreography – doesnotmatter Jul 17 '19 at 13:43

Thank you fo reply! "calling one Domain Service from another" - I mean, if I have a complex operation on entity A, then I have to use ADomainService. But this operation, in addition to entity A, affects entity B. The operation that must be performed on entity B in the ADomainService is also complex. So I have to use BDomainService from ADomainService. Now I doubt this approach :) But if I put this logic in the ApplicationService, wouldn't it break the encapsulation of business processes that should only be in the domain layer, not in application layer? – MrNVK Jul 18 '19 at 5:16

You can just emit event from your Domain Service if you think it should be in a Domain Service instead of Application Service. – doesnotmatter Jul 18 '19 at 8:07

Think a **Domain Service** as an object that implements business logic or business rules related logic on domain objects and this logic is

0

difficult to fit into the same domain objects and also doesn't cause state change of the domain service *(domain service is an object without a "state" or better without a state that has a business meaning)* but eventually change the state only of the domain objects on which operates.

While an **Application Service** implements applicative level logic as user interaction, input validation, logic not related to business but to other concerns: authentication, security, emailing, and so on.., limiting itself to simply use services exposed by domain objects.

An example of this could be the following scenario thinked only for explaining purpose: *we have to implement a very little domotic utility app that executes a simple operation, that is "turn on the lights, when someone opens the door of an house's room to enter in and turn off the light when closes the door exiting from the room".*

Simplifying a lot we consider only 2 domain entities: `Door` and `Lamp`, each of them has 2 states, respectevely `open/closed` and `on/off`, and specific methods to operate state changes on them.

In this case we need a domain service that executes the specific operation of turn on the light when someone opens the door from the outer to enter into a room, **because the door and the lamp objects cannot implement this logic in a way that we consider suited to their nature**.

We can call our domain service as `DomoticDomainService` and implement 2 methods: `OpenTheDoorAndTurnOnTheLight` and `CloseTheDoorAndTurnOffTheLight`, these 2 methods respectevely change the state of both objects `Door` and `Lamp` to `open/on` and `closed/off`.

The state of enter or exit from a room it isn't present in the domain service object and either in the domain objects, but will be implemented as simple user interaction by an application service, that we may call `HouseService`, that implements some event handlers as `onOpenRoom1DoorToEnter` and `onCloseRoom1DoorToExit`, and so on for each room *(this is only an example for explaining purpose..)*, that will respectively concern about call domain service methods to execute the attended behaviour *(we haven't considered the entity `Room` because it is only an example)*.

This example, far to be a well designed real world application, has the only purpose (as more times said) to explain what a Domain Service is and its difference from an Application Service, hope it is clear and usefull.

edited Oct 18 '19 at 8:21

answered Oct 9 '19 at 11:47

Ciro Corvino

1,528 5 13 26