DDD - Dependencies between domain model, services and repositories

Asked 10 years, 10 months ago Active 1 year, 3 months ago Viewed 16k times

16

16

Just wanted to know how others have layered their architecture. Say I have my layers as follows:

Domain Layer
--Product
--ProductService (Should the imp go into this layer?)
--IProductService
--IProductRepository

Infrastructure Layer
--ProductRepository (Imp of IProductRepository in my domain)

Now when a new product is created i have a requirement to assign a product id by calling into the ProductService.GetNextProductId() method.

Because the service has a dependency on the repository i set up the ProductService ctor with an interface of IProductRepository which can be injected later. something like this:

```
public class ProductService : IProductService
{
    private IProductRepository _repository;

    public ProductService(IProductRepository repository)
    {
        _repository = repository;
    }

    public long GetNextProductId()
    {
        return _repository.GetNextProductId();
    }
}
```

My issue is that when i use the service in the Product Class i am making reference to the Repository in the ctor when instantiating a new ProductService class. In DDD its a big no no to have such a reference. I' am not even sure if my product domain class is being set up correctly to call the service, can someone pls advise:

```
public class Product : Entity
    {
        private ProductService _svc;
        private IProductRepository _repository;

        public Product(string name, Address address) //It doesnt seem right to put parm
for IProductRepository in the ctor?
            : base(_svc.GetNextProductId) // This is where i pass the id
        {
            // where to create an instance of IProductRepository?
        }
    }
```

How can i elegantly solve this design issue? I am open to suggestions from experienced DDD'ers

EDIT:

Thanks for you comments. I also doubted if the service should be called from the Product Class. I have not used a factory pattern (yet) as the construction of the object is still simple. I dont feel it warrants a factory method yet?

I' am confused...Putting the ProductId aside if my Product class needed some other data from a Service e.g GetSystemDateTime() (i know, bad example but trying to demonstrate a non db call) where would this service method be called?

Services in DDD are logic dumps where the logic is not natrual to the domain object, right? So How does it glue together?

design-patterns architecture domain-driven-design

edited Dec 4 '18 at 6:44
Cœur
27k 14 148 205

asked Apr 16 '09 at 16:07
Th3Fix3r

Services perform tasks that the object cannot do for itself. One distinction is that services can access data stores. Your ProductId problem is a pretty good example: if ProductId were a Guid, it would be fine for Product to issue a ProductId in its constructor. In your case, you need to get ProductId from another system so you should enlist a service. – Jamie Ide Apr 17 '09 at 11:49

Yes i know i need to enlist a service to get my id, but my question is where do i call the service. Within my domain class or outside and pass it into the constructor? – Th3Fix3r Apr 17 '09 at 16:37

6 Answers

15

To your last point, services in DDD are a place to put what I describe as "awkward" logic. If you have some type of logic or work flow that has dependencies on other entities this is the type of logic that usually doesn't "fit" inside a domain object itself. Example: If I have a method on my business object to perform some type of validation, the service class might execute this method (still keeping the actual validation logic related to the entity inside its class)

Another really good example I always mention is the funds transfer method. You wouldn't have an account object transfer from one object to another, but instead you would have a service that takes the "to" account and the "from" account. Then inside the service you would invoke the withdrawal method on your "from" account and the deposit method on your "to" account. If you tried to put this inside the account entity itself it would feel awkward.

A great podcast that talks in depth about this very topic can be found here. David Laribee does a really good job explaining now only the "how" but the "why" of DDD.

answered Apr 17 '09 at 13:38
Toran Billups
26.4k 38 144 258

Thanks for the link. Will listen on my way to work on monday :p So the general consensus is that Domain Classes do not call into services, period.(?) You provided a good example of the Fund Transfer method, can u provide more details where exactly is the Transer(Account fromAccount, Account toAccount, decimal amount) method is being called? Within the domain, app, or infrastructure? – Th3Fix3r Apr 17 '09 at 16:34

It will be called within the App in response to the request to transfer the funds. – grover Apr 17 '09 at 16:59

Agreed, if you are using some presentation pattern like MVC or MVP it might be invoked inside the Presenter/Controller – Toran Billups Apr 17 '09 at 17:12

Anybody has the recording of the podcast, or knows where it is still available? DeepFriedBytes is long gone :(– Kamafeather Nov 25 '19 at 19:24

10

Your domain model shouldn't have a reference to ProductService nor to IProductRepository. If you create a new Product it has to be created through a factory - the Factory may use ProductService to get a product id.

In fact I'd wrap ProductService with an appropriate interface, such as IProductIdGeneratorService so that you can inject this into the factory using your IoC container.

answered Apr 16 '09 at 16:28
grover
2,135 1 14 15

THanks please see my comments above. – Th3Fix3r Apr 17 '09 at 1:29

You're right about your addition of services as logic dumps. These services can then be used by the application running on top of the domain model or the infrastructure running below it. In your case I think either the app, teh factory or the repository can be made responsible for determining the product id. I'm more inclined to put it into the repository. If you store an entity in the db for the first time its the repos responsibility to assign it an id, this seems to be almost the same? – grover Apr 17 '09 at 9:12

Ok so you are saying that i should generate the id outside of the domain class and pass it to the constructor as part of the creation. Whereas in my example i was trying to create the id as part of the entity which really doesn't fit? – Th3Fix3r Apr 17 '09 at 16:27

Also want to add that in my example iam not using a Factory to create my product since it fairly simple. Should i opt for a factory even for simple sceanrios? – Th3Fix3r Apr 17 '09 at 16:39

4 Yes you should really not have references to services in your entities, these services are not part of the entity state. The factory solve the problem nicely. You can read this post about service injection in entities : thinkbeforecoding.com/post/2009/03/04/… – thinkbeforecoding May 7 '09 at 14:12

3

Here is how I would structure your problem. I believe this is also the recommended DDD way of doing it.

```
public class ProductService : IProductService // Application Service class, used by
outside components like UI, WCF, HTTP Services, other Bounded Contexts, etc.
{
    private readonly IProductRepository _prodRepository;
    private readonly IStoreRepository _storeRepository;

    public ProductService(IProductRepository prodRepository, IStoreRepository
storeRepository) // Injected dependencies DI
    {
        if(prodRepository == null) throw new NullArgumentException("Prod Repo is
required."); // guard
        if(storeRepository == null) throw new NullArgumentException("Store Repo is
required."); // guard

        _prodRepository = prodRepository;
        _storeRepository = storeRepository;
    }

    public void AddProductToStore(string name, Address address, StoreId storeId) //An
exposed API method related to Product that is a part of your Application Service.
Address and StoreId are value objects.
    {
        Store store = _storeRepository.GetBy(storeId);
        IProductIdGenerator productIdGenerator = new
ProductIdGenerator(_prodRepository);
        Product product = Product.MakeNew(name, address, productIdGenerator);
    }

    ... // Rest of API
}

public class Product : Entity
{
    public static MakeNew(string name, Address address, IProductIdGenerator
productIdGenerator) // Factory to make construction behaviour more explicit
    {
        return new Product(name, address, productIdGenerator);
    }

    protected Product(string name, Address address, IProductIdGenerator
productIdGenerator)
        : base(productIdGenerator.GetNextProductId())
    {
        Name = name;
        Address = address;
    }

    ... // Rest of Product methods, properties and fields
}

public class ProductIdGenerator : IProductIdGenerator
{
    private IProductRepository _repository;

    public ProductIdGenerator(IProductRepository repository)
    {
        _repository = repository;
    }

    public long GetNextProductId()
    {
        return _repository.GetNextProductId();
    }
}

public interface IProductIdGenerator
{
    long GetNextProductId();
}
```

Basically, ProductService is part of your Application Service, that is, the entry and exit point of everything that needs to use your domain or cross its boundary. It is responsible for delegating each use cases to appropriate components that can deal with it, and coordinating between all these components if many is required to fulfill the use case.

Product is your AggregateRoot and an Entity in your Domain. It is responsible for dictating the contract of the UbiquitousLanguage that capture the Domain of your enterprise. So, in itself, it means that your domain has a concept of a Product, which contains Data and Behaviour, whichever data and behaviour you expose publicly must be a concept of the UbiquitousLanguage. It's field should not have external dependencies outside the domain model, so no services. But, it's methods can take Domain Services as parameters to help it perform behaviour logic.

ProductIdGenerator is an example of such a Domain Service. Domain Services encapsulate behaviour logic that crosses outside of an Entity's own boundary. So if you have logic that requires other aggregate roots, or external services like Repository, File System, Cryptography, etc. Basically, any logic that you can not workout from inside your Entity, without needing anything else, you might need a Domain Service. If the logic is to englobing and seems like conceptually might not really belong as a method on your Entity, it's a sign you might need an entirely new Application Service use case just for it, or you have missed a Entity in your design. It is also possible to use Domain Service from the Application Service directly, in a non double dispatch way. This is a bit similar to C# extension methods versus normal static method.

=========== To Answer your Edit questions ===============

> I also doubted if the service should be called from the Product Class.

Domain Services can be called from the product class if they are passed as a temporary reference through a method parameter. Application Services should never be called from the Product class.

> I have not used a factory pattern (yet) as the construction of the object is still simple. I dont feel it warrants a factory method yet?

It depends what you expect will take you more time, making a factory now, even though you don't have multiple construction logic, or refactoring later when you do. I think that it's not worth it for entities who do not need to be constructed in more than one way. As wikipedia explains, factory is used to make what each constructor do more explicit and differentiable. In my example, the MakeNew factory explains what this particular construction of the Entity serves as purpose: to create a new Product. You could have more factory like MakeExisting, MakeSample, MakeDeprecated, etc. Each of these factory would create a Product but for different purposes and in slightly different ways. Without a Factory, all these constructors would be named Product() and it would be hard to know which one is for what and does what. The downside is that Factory are harder to work with when you extend your entity, the child entity can't use the parent Factory to create a child, that's why I tend to do all the construction logic inside the constructors anyways, and only use Factory to have a pretty name for them.

> I' am confused...Putting the ProductId aside if my Product class needed some other data from a Service e.g GetSystemDateTime() (i know, bad example but trying to demonstrate a non db call) where would this service method be called?

Say you thought the Date implementation was a detail of the infrastructure. You would create an abstraction around it to use in your application. It would start with an interface, maybe something like IDateTimeProvider. This interface would have a method

GetSystemDateTime().

Your Application Services would be free to instantiate an IDateTimeProvider and call its methods at any time, than it could pass the result to Aggregates, Entities, Domain Services, or whatever else that would need it.

Your Domain Services would be free to hold a reference to IDateTimeProvider as a class field, but it should not create the instance itself. Either it receives it through dependency injection, or it asks for it through Service Locator.

Finally, your Entites and Aggregate Roots and Value Objects would be free to call GetSystemDateTime() and other methods of IDateTimeProvider, but not directly. It would need to go through double dispatch, where you'd give it a Domain Service as a parameter of one of it's methods, and it would use that Domain Service to query the info it wants, or perform the behaviour it needs. It could also pass itself back to the Domain Service, where the domain service would do the querying and setting.

If you consider your IDateTimeProvider to actually be a Domain Service, as a part of the Ubiquitous Language, than your Entities and Aggregate Roots can just call methods on it directly, it just can not hold a reference to it as a class field, but local variables of method parameters are fine.

> Services in DDD are logic dumps where the logic is not natrual to the domain object, right? So How does it glue together?

I think my entire answer has made this pretty clear already. Basically, you have 3 possibilities for gluing it all (that I can think of as of now at least).

1) An Application Service instantiates a Domain Service, calls a method on it, and passes the resulting return values to something else that needed it (repo, entity, aggregate root, value object, another domain service, factories, etc.).

2) A Domain Service is instantiated by the Application Domain and passed as a parameter to a method of something that will use it. Whatever uses it, does not keep a permanent reference to it, it's a local variable only.

3) A Domain Service is instantiated by the Application Domain and passed as a parameter to a method of something that will use it. Whatever uses it, uses double dispatch to use the Domain Service in a non dependent way. This means it passes to the method of the Domain Service an reference to itself, as in DomainService.DoSomething(this, name, Address).

Hope this helps. Comments are welcomed if I did anything wrong or that goes against DDD's best practices.

answered Apr 30 '14 at 20:22

Didier A.
3,058 2 28 33

1

If I understand your question correctly, you state that your Product class is calling the ProductService class. It shouldn't. You should do this in a factory class that is responsible for creating and configuring a Product. Where you call this method may also depend on when you want to issue the ProductId: We have what may be a similar case in that we need to get a number from our legacy accounting system for a project. I defer getting the number until the project is persisted so that we don't waste any numbers or have gaps. If you're in a similar situation, you may want to issue the ProductId in a repository save method instead of when the object is created.

As an aside, do you really think you'll ever have more than one ProductService or ProductRepository? If not then I wouldn't bother with the interfaces.

Edited to add:

I recommend starting small and keeping it simple by starting with two just classes, Product and ProductServices. ProductServices would perform all services, including factory and repository, since you can think of those as specialized services.

edited Apr 17 '09 at 11:53 answered Apr 16 '09 at 16:46

Jamie Ide
44.3k 14 72 115

THanks please see my comments above. – Th3Fix3r Apr 17 '09 at 1:27

0

In agreement with Marcelo, you should probably first determine if the product ID is really a domain model concept. If the business users never use or have any idea of the product ID and usually refer to the product by name, number, SKU or a natural product ID made up of name + dimensions, then this is what the domain model should be aware of.

That being said, here is how I structure my DDD projects assuming that the product ID is an auto-number field in the database:

Project.Business (Domain Model)

No references and therefore, no dependencies upon anything.

```
public class Product : Entity
{
    private Product(string name, Address address)
    {
        //set values.
    }

    //Factory method, even for simple ctor is used for encapsulation so we don't have
    //to publically expose the constructor.  What if we needed more than just a couple
    //of value objects?
    public static CreateNewProduct(string name, Address address)
    {
        return new Product(name, address);
    }

    public static GetAddress(string address, string city, string state, string zip) { }
}

public interface IProductRepository : IEnumerable<Product>
{
    void Add(Product product);
    //The following methods are extraneous, but included for completion sake.
    int IndexOf(Product product);
    Product this[int index] { get; set; }
}
```

Project.Implementation

```
public SqlProductRepository : List<ProductDataModel>, IProductRepository
{
    public SqlProductRepository(string sqlConnectionString) { }

    public void Add(Product product)
    {
        //Get new Id and save the product to the db.
    }

    public int IndexOf(Product product)
    {
        //Get the index of the base class and convert to business object.
    }

    public Product this[int index]
    {
        get { //find instance based on index and return; }
        set { //find product ID based on index and save the passed in Business object to
the database under that ID. }
    }
}
```

Project.ApplicationName (Presentation Layer)

```
public class Application
{
    IProductRepository repository = new SqlProductRepository(SqlConnectionString);

    protected void Save_Click(object sender, EventArgs e)
    {
        Product newProduct = Product.CreateNewProduct(name,
Product.GetAddress(address,city,state,zip));
        repository.Add(newProduct);
    }
}
```

If necessary, you may have:

Project.Services (Application Services Layer that uses DTOs between itself and the Presentation Layer)

answered Dec 12 '12 at 16:16
Aaron Hawkins
2,090 15 24

0

Why do you need the product id when you create the product in memory? Usually the product id is setted when you create the product in your repository.

Take a look at the following code:

var id1 = _repository.GetNextProductId(); var id2 = _repository.GetNextProductId();

Will it return two different product ids?

If the answer is yes, then it's safe (but still awkward); If the answer is no, then you'll have a huge problem;

answered Dec 12 '12 at 13:23
Marcelo Oliveira
623 5 15