SOFTWARE ENGINEERING

How accurate is “Business logic should be in a service, not in a model”?

Asked 6 years, 4 months ago Active 1 year, 1 month ago Viewed 153k times

407

402

Situation

Earlier this evening I gave an answer to a question on StackOverflow.

The question:

> Editing of an existing object should be done in repository layer or in service?
>
> For example if I have a User that has debt. I want to change his debt. Should I do it in UserRepository or in service for example BuyingService by getting an object, editing it and saving it ?

My answer:

> You should leave the responsibility of mutating an object to that same object and use the repository to retrieve this object.

Example situation:

```
class User {
    private int debt; // debt in cents
    private string name;

    // getters

    public void makePayment(int cents){
        debt -= cents;
    }
}

class UserRepository {
    public User GetUserByName(string name){
        // Get appropriate user from database
    }
}
```

A comment I received:

> Business logic should really be in a service. Not in a model.

What does the internet say?

So, this got me searching since I've never really (consciously) used a service layer. I started reading up on the Service Layer pattern and the Unit Of Work pattern but so far I can't say I'm convinced a service layer has to be used.

Take for example this article by Martin Fowler on the anti-pattern of an Anemic Domain Model:

> There are objects, many named after the nouns in the domain space, and these objects are connected with the rich relationships and structure that true domain models have. The catch comes when you look at the behavior, and you realize that there is hardly any behavior on these objects, making them little more than bags of getters and setters. Indeed often these models come with design rules that say that you are not to put any domain logic in the the domain objects. Instead there are a set of service objects which capture all the domain logic. These services live on top of the domain model and use the domain model for data.
>
> (...) The logic that should be in a domain object is domain logic - validations, calculations, business rules - whatever you like to call it.

To me, this seemed exactly what the situation was about: I advocated the manipulation of an object's data by introducing methods inside that class that do just that. However I realize that this should be a given either way, and it probably has more to do with how these methods are invoked (using a repository).

I also had the feeling that in that article (see below), a Service Layer is more considered as a façade that delegates work to the underlying model, than an actual work-intensive layer.

> Application Layer [his name for Service Layer]: Defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. The tasks this layer is responsible for are meaningful to the business or necessary for interaction with the application layers of other systems. This layer is kept thin. It does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in the next layer down. It does not have state reflecting the business situation, but it can have state that reflects the progress of a task for the user or the program.

Which is reinforced here:

> Service interfaces. Services expose a service interface to which all inbound messages are sent. You can think of a service interface as a façade that exposes the business logic implemented in the application (typically, logic in the business layer) to potential consumers.

And here:

> The service layer should be devoid of any application or business logic and should focus primarily on a few concerns. It should wrap Business Layer calls, translate your Domain in a common language that your clients can understand, and handle the communication medium between server and requesting client.

This is a serious contrast to other resources that talk about the Service Layer:

> The service layer should consist of classes with methods that are units of work with actions that belong in the same transaction.

Or the second answer to a question I've already linked:

> At some point, your application will want some business logic. Also, you might want to validate the input to make sure that there isn't something evil or nonperforming being requested. This logic belongs in your service layer.

"Solution"?

Following the guidelines in this answer, I came up with the following approach that uses a Service Layer:

```
class UserController : Controller {
    private UserService _userService;

    public UserController(UserService userService){
        _userService = userService;
    }

    public ActionResult MakeHimPay(string username, int amount) {
        _userService.MakeHimPay(username, amount);
        return RedirectToAction("ShowUserOverview");
    }

    public ActionResult ShowUserOverview() {
        return View();
    }
}

class UserService {
    private IUserRepository _userRepository;

    public UserService(IUserRepository userRepository) {
        _userRepository = userRepository;
    }

    public void MakeHimPay(username, amount) {
        _userRepository.GetUserByName(username).makePayment(amount);
    }
}

class UserRepository {
    public User GetUserByName(string name){
        // Get appropriate user from database
    }
}

class User {
    private int debt; // debt in cents
    private string name;

    // getters

    public void makePayment(int cents){
        debt -= cents;
    }
}
```

Conclusion

All together not much has changed here: code from the controller has moved to the service layer (which is a good thing, so there is an upside to this approach). However this doesn't look like it had anything to do with my original answer.

I realize design patterns are guidelines, not rules set in stone to be implemented whenever possible. Yet I have not found a definitive explanation of the service layer and how it should be regarded.

- Is it a means to simply extract logic from the controller and put it inside a service instead?
- Is it supposed to form a contract between the controller and the domain?
- Should there be a layer between the domain and the service layer?

And, last but not least: following the original comment

> Business logic should really be in a service. Not in a model.

- Is this correct?
  - How would I introduce my business logic in a service instead of the model?

design-patterns layers

edited Feb 14 '19 at 20:06
Dave Cousineau
189 ●1 ●8

asked Nov 9 '13 at 21:54
Jeroen Vannevel
5,217 ●6 ●15 ●27

6 I treat service layer as the place where to put the unavoidable part of transaction script acting upon aggregate roots. If my service layer becomes too complex that signals me that I move in direction of Anemic Model and my domain model needs attention and review. I also try to put the logic in SL which is not going to be duplicated by its nature. – Pavel Voronin Nov 12 '13 at 16:35

I thought that services were part of the Model layer. Am I wrong in thinking that? – Florian Margaine Nov 13 '13 at 10:10

I use a rule of thumb: don't depend on what you don't need; otherwise I may be (usually negatively) impacted by changes on the part I don't need. As a consequence, I end up with lots of clearly defined roles. My data objects contain behavior as long as all clients use it. Otherwise I move the behavior into classes implementing the required role. – beluchin Jun 30 '15 at 15:12

1 Application flow control logic belongs in a controller. Data access logic belongs in a repository. Validation logic belongs in a service layer. A service layer is an additional layer in an ASP.NET MVC application that mediates communication between a controller and repository layer. The service layer contains business validation logic. repository.asp.net/mvc/overview/older-versions-1/models-data/… – Kbdavis07 Jul 14 '16 at 10:03

3 IMHO, correct OOP-style domain model should indeed avoid "business services" and keep the business logic in the model. But practice shows that it's so tempting to create a huge business layer with distinctly named methods and also to put a transaction (or unit of work) around entire method. It's much easier to process multiple model classes in one business service method than to plan relations between those model classes, because then

you would have some tough questions to answer: where is aggregate root? Where should I start/commit a database transaction? Etc. – JustAMartin Apr 21 '17 at 9:52

12 Answers

Active Oldest Votes

384

+50

In order to define what a *service's* responsibilities are, you first need to define what a *service* is.

Service is not a canonical or generic software term. In fact, the suffix `Service` on a class name is a lot like the much-maligned Manager: It tells you almost nothing about what the object actually *does*.

In reality, what a service ought to do is highly architecture-specific:

1. In a traditional layered architecture, *service* is literally synonymous with *business logic layer*. It's the layer between UI and Data. Therefore, **all** business rules go into services. The data layer should only understand basic CRUD operations, and the UI layer should deal only with the mapping of presentation DTOs to and from the business objects.
2. In an RPC-style distributed architecture (SOAP, UDDI, BPEL, etc.), the *service* is the logical version of a physical *endpoint*. It is essentially a collection of operations that the maintainer wishes to provide as a public API. Various best practices guides explain that a service *operation* should in fact be a business-level operation and not CRUD, and I tend to agree.

   However, because routing *everything* through an actual remote service can seriously hurt performance, it's normally best *not* to have these services actually implement the business logic themselves; instead, they should wrap an "internal" set of business objects. A single service might involve one or several business objects.
3. In an MVP/MVC/MVVM/MV* architecture, *services* don't exist at all. Or if they do, the term is used to refer to any generic object that can be injected into a controller or view model. The business logic is in your *model*. If you want to create "service objects" to orchestrate complicated operations, that's seen as an implementation detail. A lot of people, sadly, implement MVC like this, but it's considered an anti-pattern (Anemic Domain Model) because the model itself does nothing, it's just a bunch of properties for the UI.

   Some people mistakenly think that taking a 100-line controller method and shoving it all into a service somehow makes for a better architecture. It really doesn't; all it does is add another, probably unnecessary layer of indirection. *Practically* speaking, the controller is still doing the work, it's just doing so through a poorly named "helper" object. I highly recommend Jimmy Bogard's Wicked Domain Models presentation for a clear example of how to turn an anemic domain model into a useful one. It involves careful examination of the models you're exposing and which operations are actually valid in a *business* context.

   For example, if your database contains Orders, and you have a column for Total Amount, your application *probably* shouldn't be allowed to actually change that field to an arbitrary value, because (a) it's history and (b) it's supposed to be determined by what's *in* the order as well as perhaps some other time-sensitive data/rules. Creating a service to manage Orders does not necessarily solve this problem, because user code can *still* grab the actual Order object and change the amount on it. Instead, the order *itself* should be responsible for ensuring that it can only be altered in safe and consistent ways.

4. In DDD, services are meant specifically for the situation [when you have an operation that doesn't properly belong to any aggregate root](). You have to be careful here, because often the need for a service can imply that you didn't use the correct roots. But assuming you did, a service is used to coordinate operations across multiple roots, or sometimes to handle concerns that don't involve the domain model at all (such as, perhaps, writing information to a BI/OLAP database).

   One notable aspect of the DDD service is that it is allowed to use [transaction scripts](). When working on large applications, you're very likely to eventually run into instances where it's just way easier to accomplish something with a T-SQL or PL/SQL procedure than it is to fuss with the domain model. This is OK, and it belongs in a Service.

   This is a radical departure from the layered-architecture definition of services. A service layer encapsulates domain objects; a DDD service encapsulates whatever **isn't** in the domain objects and doesn't make sense to be.

5. In a Service-Oriented Architecture, a *service* is considered to be the technical authority for a business capability. That means that it is the *exclusive* owner of a certain subset of the business data and nothing else is allowed to touch that data - not even to just *read* it.

   By necessity, services are actually an end-to-end proposition in an SOA. Meaning, a service isn't so much a specific *component* as an entire *stack*, and your entire application (or your entire business) is a set of these services running side-by-side with no intersection except at the messaging and UI layers. Each service has its own data, its own business rules, and its own UI. They don't need to orchestrate with each other because they are supposed to be business-aligned - and, like the business itself, each service has its own set of responsibilities and operates more or less independently of the others.

   So, by the SOA definition, every piece of business logic *anywhere* is contained within the service, but then again, so is the entire *system*. Services in an SOA can have *components*, and they can have *endpoints*, but it's fairly dangerous to call any piece of code a *service* because it conflicts with what the original "S" is supposed to mean.

   Since SOA is generally pretty keen on messaging, the operations that you might have packaged in a *service* before are generally encapsulated in *handlers*, but the multiplicity is different. Each handler handles *one* message type, *one* operation. It's a strict interpretation of the [Single Responsibility Principle](), but makes for great maintainability because every possible operation is in its own class. So you don't really *need* centralized business logic, because commands represents business operations rather than technical ones.

Ultimately, in any architecture you choose, there is going to be some component or layer that has most of the business logic. After all, if business logic is scattered all over the place then you just have spaghetti code. But whether or not you call that component a *service*, and how it's designed in terms of things like number or size of operations, depends on your architectural goals.

There's no right or wrong answer, only what applies to your situation.

answered Nov 14 '13 at 3:15

Aaronaught
40.7k ● 10 ● 85 ● 124

13 Thank you for the very elaborate answer, you have clarified all I can think of. While the other answers are good to excellent quality as well, I believe this answer tops them all thus I will accept this one. I'll add it here for the other answers: exquisite quality and information, but sadly I'll only be able to give you an upvote. – Jeroen Vannevel Nov 14 '13 at 4:19

2 I don't agree with the fact that in a traditional layered architecture, service is a synonym to the business logic layer. – CodeART Nov 14 '13 at 8:46

1 @CodeART: It is in a 3-tier architecture. I *have* seen 4-tier architectures where there is an "application layer" between the presentation and business layers, which is sometimes also called a "service" layer, but honestly, the only places I've ever seen this implemented *successfully* are huge sprawling infinitely-configurable run-your-whole-business-for-you products from the likes of SAP or Oracle, and I didn't think it was really worth mentioning here. I can add a clarification if you like. – Aaronaught Nov 14 '13 at 12:36

1 But if we take 100+ line controller (for example that controller accept message - then deserialize JSON object, make validation, apply business rules, save to db , return result object) and move some logic to one of the that called service method doesn't that help us separately unit test each part of it painlessly ? – artjom Jan 6 '14 at 9:54

2 @Aaronaught I wanted to clarify one thing if we have domain objects mapped to db via ORM and there are no business logic in them is this anemic domain model or no ? – artjom Jan 7 '14 at 8:06

41

As for your **title**, I don't think the question makes sense. The MVC Model consists of data and business logic. To say logic should be in the Service and not the Model is like saying, "The passenger should sit in the seat, not in the car".

Then again, the term "Model" is an overloaded term. Perhaps you didn't mean MVC Model but you meant model in the Data Transfer Object (DTO) sense. AKA an Entity. This is what Martin Fowler is talking about.

The way I see it, Martin Fowler is speaking of things in an ideal world. In the real world of Hibernate and JPA (in Java land) the DTOs are a super leaky abstraction. I'd love to put my business logic in my entity. It'd make things way cleaner. The problem is these entities can exist in a managed/cached state that is very difficult to understand and constantly prevents your efforts. To summarize my opinion: Martin Fowler is recommending the right way, but the ORMs prevent you from doing it.

I think Bob Martin has a more realistic suggestion and [he gives it in this video that's not free](). He talks about keeping your DTOs free of logic. They simply hold the data and transfer it to another layer that's much more object oriented and doesn't use the DTOs directly.

This avoids the leaky abstraction from biting you. The layer with the DTOs and the DTOs themselves are not OO. But once you get out of that layer, you get to be as OO as Martin Fowler advocates.

The benefit to this separation is it abstracts away the persistence layer. You could switch from JPA to JDBC (or vice versa) and none of the business logic would have to change. It just depends on the DTOs, it doesn't care *how* those DTOs get populated.

To *slightly* change topics, you need to consider the fact that SQL databases aren't object oriented. But ORMs usually have an entity - which is an object - per table. So right from the start you've already lost a battle. In my experience, you can never represent the Entity the exact way you want in an object oriented way.

As for "**a** service", Bob Martin would be against having a class named `FooBarService`. That's not object oriented. What does a service do? *Anything* related to `FooBars`. It may as well be labeled `FooBarUtils`. I think he'd advocate a service layer (a better name would be the business logic layer) but every class in that layer would have a meaningful name.

edited Nov 12 '13 at 22:44

answered Nov 12 '13 at 22:29
Daniel Kaplan
6,188 ●5 ●27 ●43

2 Agree with your point on ORMs; they propagate a lie that you map your entity directly to the db with them, when in reality an entity may be stored across several tables. – Andy Nov 20 '14 at 1:29

@Daniel Kaplan do you know what the updated link is for the Bob Martin video? – Brian Morearty Jan 18 '18 at 22:29

26

I'm working on the greenfield project right now and we had to make few architectural decisions just yesterday. Funnily enough I had to revisit few chapters of 'Patterns of Enterprise Application Architecture'.

This is what we came up with:

- Data layer. Queries and updates database. The layer is exposed through injectable repositories.
- Domain layer. This is where the business logic lives. This layer makes a use of injectable repositories and is responsible for the majority of business logic. This is the core of the application that we will test thoroughly.
- Service layer. This layer talks to the domain layer and serves the client requests. In our case the service layer is quite simple - it relays requests to the domain layer, handles security and few other cross cutting concerns. This is not much different to a controller in MVC application - controllers are small and simple.
- Client layer. Talks to the service layer through SOAP.

We end up with the following:

Client -> Service -> Domain -> Data

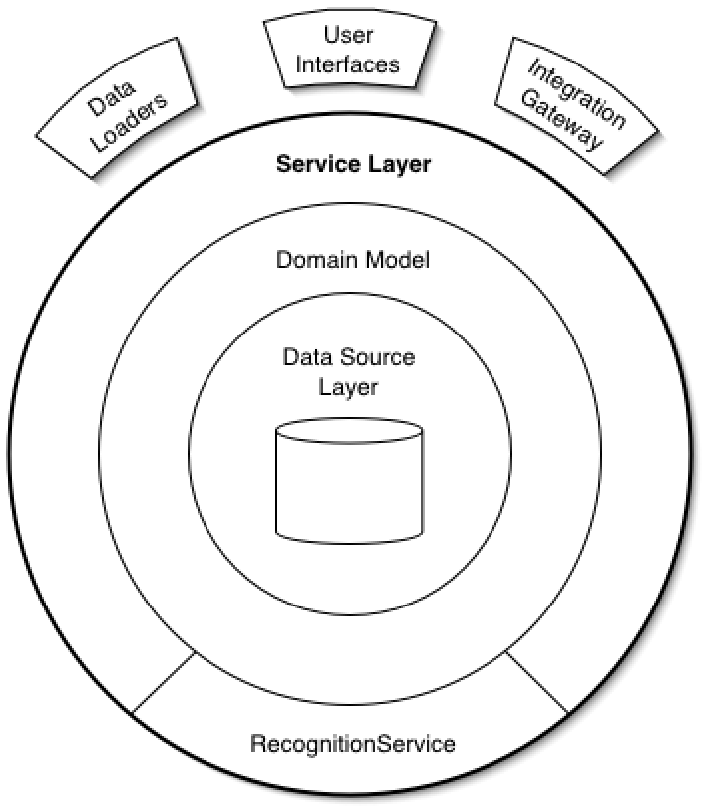
We can replace client, service or data layer with reasonable amount of work. If your domain logic lived in the service and you've decided that you want to replace or even remove your service layer, then you'd have to move all of the business logic somewhere else. Such a requirement is rare, but it might happen.

Having said all this, I think this is pretty close to what Martin Fowler meant by saying

> These services live on top of the domain model and use the domain model for data.

Diagram below illustrates this pretty well:

http://martinfowler.com/eaaCatalog/serviceLayer.html



edited Nov 14 '13 at 8:23 answered Nov 13 '13 at 20:37

gnat
21.5k ● 19 ● 96 ● 240

CodeART
3,744 ● 1 ● 16 ● 23

2 You decided for SOAP just yesterday? Is that a requirement or did you just have no better idea? – JensG Nov 14 '13 at 0:15

1 REST won't cut it for our requirements. SOAP or REST, this doesn't make any difference to the answer. From my understanding, service is a gateway to domain logic. – CodeART Nov 14 '13 at 8:17

Absolutely agree with the Gateway. SOAP is (standardized) bloatware, so I had to ask. And yes, no impact on question/answer either. – JensG Nov 14 '13 at 17:44

6 Do yourself a favor and kill your service layer. Your ui should use your domain directly. Ive seen this before and your domain invariably becomes a bunch of anemic dtos, not rich models. – Andy Nov 20 '14 at 1:32

Those slides cover the explanation regarding service layer and this graphic above pretty neat: slideshare.net/ShwetaGhate2/… – Marc Juchli Jan 9 '15 at 12:10

9

This is one of those things that really depends on the use case. The overall point of a service layer is to consolidate business logic together. This means that several controllers can call the same UserService.MakeHimPay() without actually caring about how the payment is done. What goes on in the service may be as simple as modifying an object property or it may be doing complex logic dealing with other services (i.e., calling out to third party services, calling validation logic, or even just simply saving something to the database.)

This doesn't mean you have to strip ALL logic from the domain objects. Sometimes it just makes more sense to have a method on the domain object do some calculations on itself. In your final example, the service is a redundant layer over the repository/domain object. It provides a nice buffer against requirements changes, but it really isn't necessary. If you think you need a service, try having that do the simple "modify property X on object Y" logic instead of the domain object. Logic on the domain classes tends to fall into the "calculate this value from fields" rather than exposing all fields via getters/setters.

answered Nov 10 '13 at 0:55

firelore
444 ● 2 ● 2

2 Your stance in favor of a service layer with business logic makes a lot of sense, but this still leaves some questions. In my post I have quoted several respectable sources which talk about the service layer as a façade void of any business logic. This is in direct contrast with your answer, could you perhaps clarify this difference? – Jeroen Vannevel Nov 11 '13 at 5:18

5 I find that it really depends on the TYPE of business logic and other factors, such as the language being used. Some business logic doesn't fit on the domain objects very well. One example is filtering/sorting results after pulling them back from the database. It IS business logic, but it doesn't make

sense on the domain object. I find that services are best-used for simple logic or transforming the results and logic on the domain is most useful when it deals with saving data or calculating data from the object. – firelore Nov 12 '13 at 4:06

9

I think the answer is clear if you read Martin Fowler's Anemic Domain Model article.

Removing the business logic, which is the domain, from the domain model is essentially breaking object oriented design.

Let's review the most basic object oriented concept: An object encapsulates data and operations. For instance, closing an account is an operation that an account object should perform on itself; therefore, having a service layer perform that operation is not an object oriented solution. It is procedural, and it is what Martin Fowler is referring to when he is talking about an anemic domain model.

If you have a service layer close the account, rather than having the account object close itself, you don't have a real account object. Your account "object" is merely a data structure. What you end up with, as Martin Fowler suggests, is a bunch of bags with getters and setters.

edited Jun 22 '16 at 17:16
BadHorsie
133 ● 6

answered Apr 9 '15 at 12:28
Carlos A Merighe - Utah
101 ● 1 ● 1

1 Edited. I actually found this quite a helpful explanation and don't think it deserves downvotes. – BadHorsie Jun 22 '16 at 14:59

1 There is one largely overseen downside of rich models. And that is developers pulling in anything slightly related into the model. Is the state of open/closed an attribute of the account? What about the owner? And the bank? Should they all be referenced by the account? Would I close an account by talking to a bank or through the account directly? With anemic models, those connections are not an inherent part of the models, but are rather created when working with those models in other classes (call them services, or managers). – Hubert Grzeskowiak May 23 '19 at 8:33

8

The easiest way to illustrate why programmers shy away from putting domain logic in the domain objects is that they're usually faced with a situation of "where do I put the validation logic?" Take this domain object for instance:

```
public class MyEntity
{
    private int someProperty = 0;

    public int SomeProperty
    {
        get { return this.someProperty; }
        set
        {
            if(value < 0) throw new ArgumentOutOfRangeException("value");
            this.someProperty = value;
        }
    }
}
```

So we have some basic validation logic in the setter (cannot be negative). The problem is that you can't really re-use this logic. Somewhere there's a screen or a ViewModel or a Controller that needs to do validation *before* it actually commits the change to the domain object, because it needs to inform the user either before or when they click the Save button that they can't do that, and *why*. Testing for an exception when you call the setter is an ugly hack because you really should have done all the validation before you even started the transaction.

That's why people move the validation logic some kind of service, such as `MyEntityValidator`. Then the entity and the calling logic can both get a reference to the validation service and re-use it.

If you *don't* do that and you still want to re-use validation logic, you end up putting it in static methods of the entity class:

```
public class MyEntity
{
    private int someProperty = 0;

    public int SomeProperty
    {
        get { return this.someProperty; }
        set
        {
            string message;
            if(!TryValidateSomeProperty(value, out message))
            {
                throw new ArgumentOutOfRangeException("value", message);
            }
            this.someProperty = value;
        }
    }

    public static bool TryValidateSomeProperty(int value, out string message)
    {
        if(value < 0)
        {
            message = "Some Property cannot be negative.";
            return false;
        }
        message = string.Empty;
        return true;
    }
}
```

This would make your domain model less "anemic", and keep the validation logic next to the property, which is great, but I don't think anyone really likes the static methods.

answered Sep 11 '14 at 14:47
Scott Whitlock
21.2k ● 4 ● 54 ● 85

1 So what is the best solution for validation in your opinion? – Flashrunner Sep 5 '16 at 16:26

3 @Flashrunner - my validation logic is definitively in the business logic layer, but is also duplicated in the entity and database layers in some cases. The business layer handles it nicely by informing the user, etc., but the other layers just throw errors/exceptions and keep the programmer (myself) from making mistakes that corrupt data. – Scott Whitlock Sep 5 '16 at 16:41

4

How would you implement your business logic in the service layer? When you make a payment from a user, you a creating a payment, not just deducting a value from a property.

Your make payment method needs to create a payment record, add to that user's debt and persist all of this in your repositories. Doing this in a service method is incredibly straightforward, and you can also wrap the whole operation in a transaction. Doing the same in an aggregated domain model is much more problematic.

answered Nov 9 '13 at 23:57
Mr Cochese
1,248 ● 9 ● 8

He is not creating a payment record though. That's why in his case this business logic should live in the entity. But if he had to create a payment record - you are right, the payment business logic would live elsewhere. – anton1980 Feb 12 at 3:48

2

The tl;dr version:
My experiences and opinions say that any objects which have business logic should be part of the domain model. The data model should likely not have any logic whatsoever. Services should likely tie the two together, and deal with cross-cutting concerns (databases, logging, etc.). However, the accepted answer is the most practical one.

The longer version, which has kind of been alluded to by others, is that there is an equivocation on the word "model". The post switches between data model and domain model as if they're the same, which is a very common mistake. There may also be a slight equivocation on the word "service".

In practical terms, you should not have a service which makes changes to any domain objects; the reason for this is that your service will likely have some method for every property on your object in order to change the value of that property. This is a problem because then, if you have an interface for your object (or even if not), the service is no longer following the Open-Closed Principle; instead, whenever you add more data to your model (regardless of domain vs data), you end up having to add more functions to your service. There are certain ways around it, but this is the most common reason I've seen "enterprise" applications fail, especially when those organizations think that "enterprise" means "have an interface for every object in the system". Can you imagine adding new methods to an interface, then to two or three different implementations (the in-app one, the mock implementation, and debug one, the in-memory one?), just for a single property on your model? Sounds like a terrible idea to me.

There's a longer problem here that I won't go into, but the gist is this: Hardcore object-oriented programming says that no one outside of the relevant object should be able to change the value of a property within the object, nor even "see" the value of the property within the object. This can be alleviated by making the data read-only. You can still run into issues like when a lot of people make use of the data even as read-only and you have to change the type of that data. It's possible that all the consumers will have to change to accommodate that. This is why, when you make APIs to be consumed by anyone and everyone, you're advised not to have public or even protected properties/data; it's the very reason OOP was invented, ultimately.

I think the majority of the answers here, aside from the one marked accepted, are all clouding the issue. The one marked accepted is good, but I still felt the need to reply and agree that bullet 4 is the way to go, in general.

> In DDD, services are meant specifically for the situation when you have an operation that doesn't properly belong to any aggregate root. You have to be careful here, because often the need for a service can imply that you didn't use the correct roots. But assuming you did, a service is used to coordinate operations across multiple roots, or sometimes to handle concerns that don't involve the domain model at all...

edited Apr 12 '17 at 7:31
Community ♦
1

answered Nov 15 '13 at 19:52
WolfgangSenff
261 ● 1 ● 4

1

The answer is that it depends on the use case. But in most generic scenarios, I would adhere to business logic laying in the service layer. The example that you have provided is a really simple one. However once you start thinking of decoupled systems or services and adding transactional behaviour on top of it you really want it to happen as part of the service layer.

The sources you have quoted for service layer being devoid of any business logic introduces another layer which is the business layer. In many scenarios the service layer and business layer are compressed into one. It really depends on how you want to design your system. You can get the job done in three layers and keep decorating on and adding noise.

What you can ideally do is ***model services which encompass business logic to work on domain models to persist state***. You should try to decouple services as much as possible.

edited Nov 13 '13 at 8:23
gnat
21.5k ● 19 ● 96 ● 240

answered Nov 13 '13 at 8:17
sunny
137 ● 5

0

In MVC Model is defined as the business logic. Claiming it should be somewhere else is incorrect unless he is not using MVC. I view service layers as similar to a module system. It allows you to bundle up a set of related functionality into a nice package. The internals of that service layer would have a model doing the same work as yours.

> The model consists of application data, business rules, logic, and functions.
> http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

answered Nov 12 '13 at 21:37
stonemetal
3,331 ● 13 ● 17

0

The concept of Service layer can be viewed from DDD perspective. Aaronaught mentioned it in his answer, I just elaborate on it a bit.

Common approach there is to have a controller that is specific to a type of client. Say, it could be a web-browser, it could be some other application, it could be a functional test. The request and response formats might vary. So I use application service as a tool for utilizing a Hexagonal architecture. I inject there infrastructure classes specific to a concrete request. For example, that's how my controller serving web-browser requests could look like:

```
class WebBroserController
{
    public function purchaseOrder()
    {
        $data = $_POST;

        $responseData =
            new PurchaseOrderApplicationService(
                new PayPalClient(),
                new OrderRepository()
            )
        ;
        $response = new HtmlView($responseData);
    }
}
```

If I'm writing a functional test, I want to use a fake payment client and probably I wouldn't need an html response. So my controller could look like that:

```
class FunctionalTestController
{
    public function purchaseOrder()
    {
        $data = $_POST;

        $responseData =
            new PurchaseOrderApplicationService(
                new FakePayPalClient(),
                new OrderRepository(),
                $data
            )
        ;

        return new JsonData($responseData);
    }
}
```

So application service is an environment that I set up for running business-logic. It's where the model classes are called -- agnostically of an infrastructure implementation.

So, answering your questions from this perspective:

> Is it a means to simply extract logic from the controller and put it inside a service instead?

No.

> Is it supposed to form a contract between the controller and the domain?

Well, one can call it so.

> Should there be a layer between the domain and the service layer?

Nope.

There is a radically different approach though that totally denies the use of any kind of service. For example, David West in his book Object Thinking claims that any object should have all necessary resources to do its job. This approach results in, for example, discarding any ORM.

answered Nov 3 '17 at 13:19
Zapadlo
1,790 ● 1 ● 10 ● 15

-2

For the record.

SRP:

1. Model = Data, here goes the setter and getters.
2. Logic/Services = here goes the decisions.
3. Repository/DAO = here we permantenty store or retrieve the information.

In this case, is OK to do the next steps:

If the debt will not require some calculation:

```
userObject.Debt = 9999;
```

However, if it requires some calculation then:

```
userObject.Debt= UserService.CalculateDebt(userObject)
```

or also

```
UserService.UpdateDebt(userObject)
```

But also, if the calculation is done in the persistence layer, such a store procedure then

```
UserRepository.UpdateDebt(userObject)
```

In this case, i we want to retrieve the user from the database and to update the debt then, we should do it in several step (in fact, two) and it's not requires to wrap/encapsulate it in a service's function.

```
User userObject=UserRepository.GetUserByName(somename);
UserService.UpdateDebt(userObject)
```

And if it requires to store it then, we can add a third step

```
User userObject=UserRepository.GetUserByName(somename);
UserService.UpdateDebt(userObject)
UserRepository.Save(userobject);
```

About the solution proposed

a) We shouldn't be afraid to left the end-developer to write a couple of instead of encapsulate it in a function.

b) And about Interface, some developers love interface and they are fine but in several cases, they aren't need at all.

c) The goal of a service is to create one without attributes, mainly because we can use Shared/Static functions. It's also easy to unit test.

answered Sep 11 '14 at 12:28
magallanes
208 ● 1 ● 5

how does this answer the question asked: *How accurate is “Business logic should be in a service, not in a model”?* – gnat Sep 11 '14 at 12:36

3 What kind of Sentence is `"We shouldn't be afraid to left the end-developer to write a couple of instead of encapsulate it in a function. `"? I can only quote Lewis Black "if it weren't for my horse I wouldn't have spent that year in college". – Malachi Oct 7 '14 at 21:42