Separating business rules from entities in domain driven design

Asked 2 years, 2 months ago Active 2 years, 2 months ago Viewed 1k times

1

2

While i am practicing DDD in my software projects, i have always faced the question of "Why should i implement my business rules in the entities? aren't they supposed to be pure data models?"

Note that, from my understanding of DDD, domain models could be consist of persistent models as well as value objects.

I have come up with a solution in which i separate my persistent models from my domain models. On the other hand we have data transfer objects (DTO), so we have 3 layers of data mapping. Database to persistence model, persistence model to domain models and domain models to DTOs. In my opinion, my solution is not an efficient one as too much hard effort must be put into it.

Therefore is there any better practice to achieve this goal?

domain-driven-design software-design

asked Jan 21 '18 at 9:20
Behzad
727 ●3 ●24

The question of too much layering/mapping has been discussed hundreds of times on SO. It is also completely unrelated to "implementing business rules in the entities". – guillaume31 Jan 22 '18 at 14:38

@guillaume31 Could you provide some links? Perhaps those could help Behzad... – Mark Seemann Jan 22 '18 at 15:04

2 @MarkSeemann Do you have this one in mind? – guillaume31 Jan 22 '18 at 15:20

1 @guillaume31 I had none in mind, and while I agree with you that this is a topic basically done to death here, we can't expect newcomers to know this... and I wasn't fishing for links to my blog, but thanks :) – Mark Seemann Jan 22 '18 at 15:36

1 @guillaume31 i am aware of object-object mappers/layering and my question was not about it. I have searched several times in SO and i couldn't find my answer but in case if i made a mistake i would appreciate your help. Anyway, Entirely different question might have the same answer and it is good for the community that reach to the answer from different perspectives. – Behzad Jan 22 '18 at 16:45

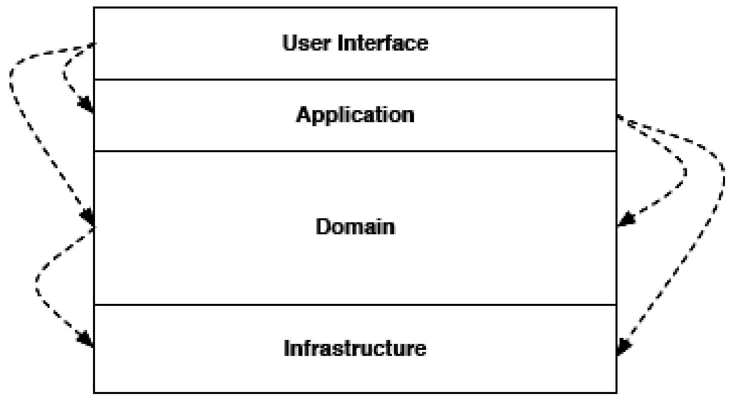
3

Disclaimer: this answer is a little larger that the question but it is needed to understand the problem; also is 100% based on my experience.

What you are feeling is normal, I had the same feeling some time ago. This is because of a combination of architecture, programming language and used framework. You should try to choose the above tools as such that they give the code that is easiest to change. If you have to change 3 classes for each field added to an entity then this would be nightmare in a large project (i.e. 50+ entity types).

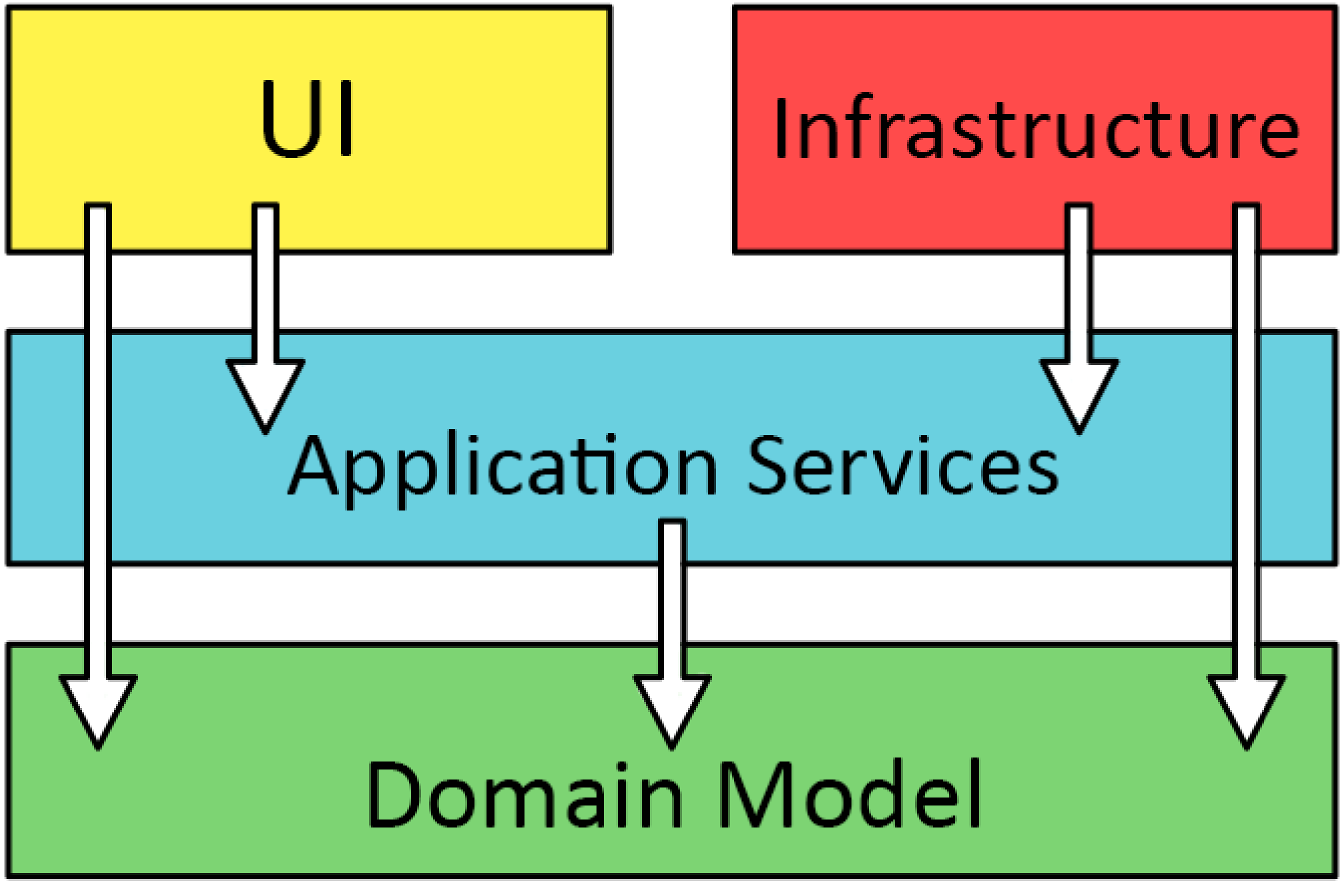
The problem is that you have multiple DTOs per entity/concept.

The heaviest architecture that I used was the Classic layered architecture; the strict version was the hardest (in the strict version a layer may access only the layer that is just before it; i.e. the User interface may access only the Application). It involved a lot of DTOs and translations as the data moved from the Infrastructure to the UI. The testing was also hard as I had to use a lot of mocking.

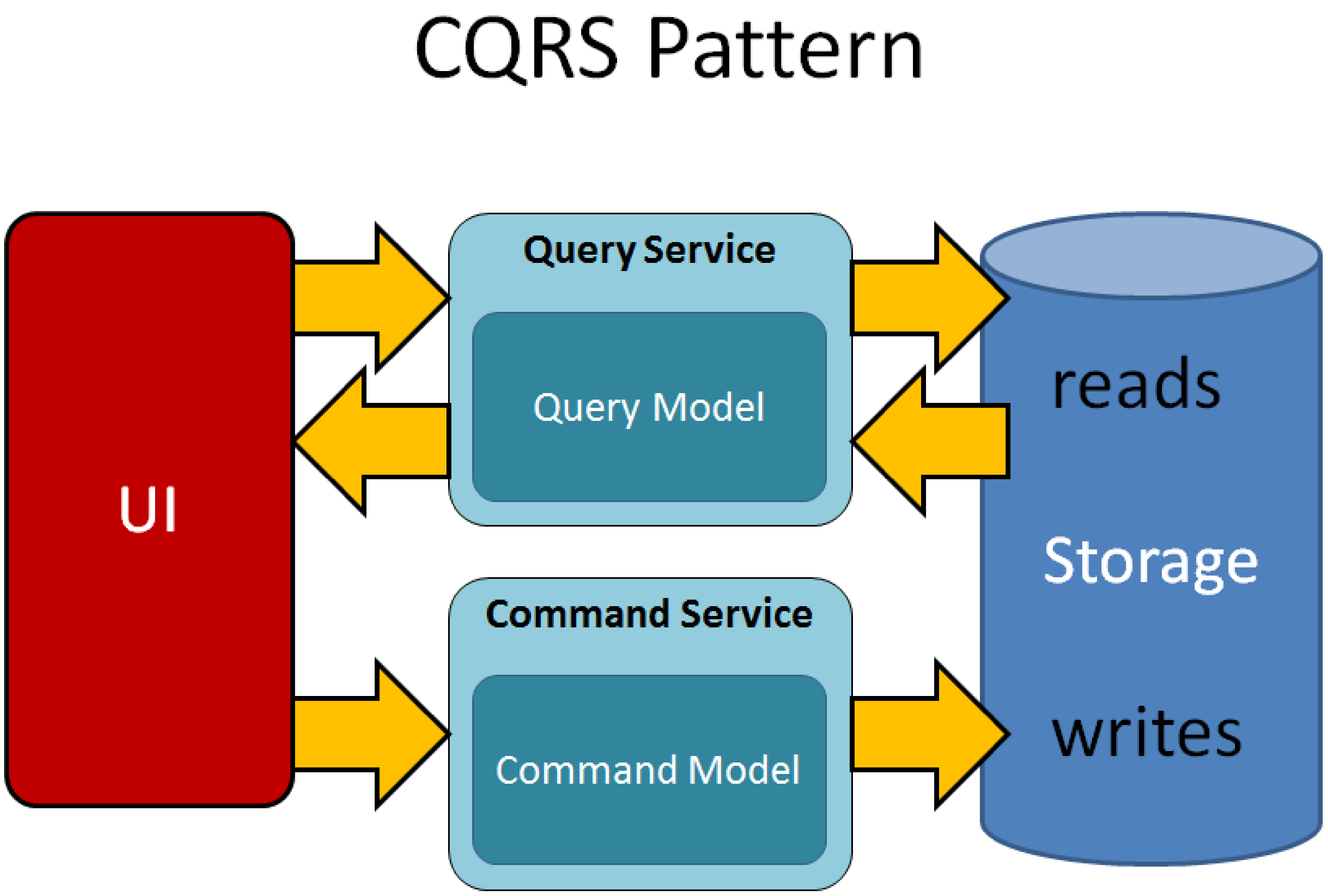


Then I inverted the dependency, the Domain will not depend on the Infrastructure. For this I defined interfaces in the Domain layer that were implemented in the Infrastructure. But I still needed to use mocking for them. Also, the Aggregates were not pure and they had side effects (because they called the Infrastructure, even it was abstracted by interfaces).

Then I moved the Domain to the very bottom. This made my Aggregates pure. I no longer needed to use mocking. But I still needed DTOs (returned by the Application layer to the UI and those used by the ORM).



Then I made the first leap: CQRS. This splits the models in two: the write model and the read model. The important thing is that you don't need to use DTOs for models anymore. The Aggregate (the write model) can be serialized as it is or converted to JSON and stored in almost any database. Vaughn Vernon has a blog post about this.



But the nicest are the Read models. You can create a read model for each use case. Being a model used only for read/query, it can be as simple/dump as possible. The read entities contain only query related behavior. With the right persistence they can be persisted as they are. For example, if you use `MongoDB` (or any document database), with a simple reflection based serializer you can have a very

The second leap is Event sourcing. With this you don't need a flat persistence for the Aggregates. They are rehydrated from the Event store each time they handle a command.

You still have DTOs (commands, events, read models) but there is only one DTO per entity/concept.

Regarding the elimination of DTOs used by the Presentation: you can use something like GraphSQL.

All the above can be made worse by the programming language and framework. Strong typed programming languages force you to create a type for each custom returned value. Some frameworks force you to return a custom serializable type in order to return them to REST over HTTP requests (in this way you could have self-described REST endpoints using reflection). In `PHP` you can simply use arrays with string keys as value to be returned by a REST controller.

P.S.

- By DTO I mean a class with data and no behavior.
- I'm not saying that we all should use CQRS, just that you should know that it exists.

answered Jan 22 '18 at 8:48
Constantin Galbenu
12.9k ● 3 ● 18 ● 35

You have answered my question. but one more thing, for the sake of isolation and security, exposing my aggregate models sounds fishy. or am i missing something? – Behzad Jan 22 '18 at 14:45

@Behzad In which of the architectures and for what purpose: read or write? – Constantin Galbenu Jan 22 '18 at 15:27

for write purposes in CQRS. What i understand is we have different commands which they are responsible to alter the state of each aggregate without the use of DTOs which means exposing the data as it is. – Behzad Jan 22 '18 at 16:34

@Behzad The commands are just DTOs (structure with no behavior other than primitive validation). They are sent to a command handler (an Application service) that loads the Aggregate from the Persistence (in case of Event sourcing: from the Event store), it calls the appropriate method on the Aggregate, it collects the generated events (other DTOs) and then it persist the Aggregate and the events (in case of Event sourcing it persists only the events). So, the Aggregate is *exposed* (=used) only by the Command handler. – Constantin Galbenu Jan 22 '18 at 17:44

@Behzad I use another style, the one promoted by cqrs.nu in which the command arrives directly at the Aggregate; the command handler is very generic. I like this style as it reduces a lot the number of Application command handlers to one, with just a little convention: aggregate's command methods have the format `handleXXX` where XXX is the command's short name. In my latest project (a custom `CRM` with ~100 entity types) I have almost zero class duplication. Imagine what it would mean to have N x 100 classes instead of 100. – Constantin Galbenu Jan 22 '18 at 17:49

2

> Why should i implement my business rules in the entities? aren't they supposed to be pure data models?

Your persistence entities should be pure data models. Your *domain* entities describe behaviors. They aren't the same thing; it is a common pattern to have a bit of logic with in the repository to change one to the other.

The cleanest way I know of to manage things is to treat the persistent entity as a *value object* to be managed by the domain entity, and to use something like a data mapper for transitions between domain and persistence.

> On the other hand we have data transfer objects (DTO), so we have 3 layers of data mapping. Database to persistence model, persistence model to domain models and domain models to DTOs. In my opinion, my solution is not an efficient one as too much hard effort must be put into it.

cqrs offers some simplification here, based on the idea that if you are implementing a query, you don't really need the "domain model" because you aren't actually going to change the supporting data. In which case, you can take the "domain model" out of the loop altogether.

answered Jan 21 '18 at 14:08
VoiceOfUnreason
29.7k ● 3 ● 25 ● 56

2

DDD and data are very different things. The aggregate's data (an outcome) will be persisted somehow depending on what you're using. Personally I think in domain events so the resulting Domain Event is the DTO (technically it is) that can be stored directly in an Event Store (if you're using Event Sourcing) or act as a data source for your persistence model.

A domain model represents *relevant* domain behaviour with the domain state being the 'result'. An entity is concept which has an id, compared to a Value Object which represents a business semantic value only. An entity usually groups related value objects and consistency rules. Not all business rules are here , some of them make sense as a service.

Always think of DDD as a methodology to gather requirements and to structure information. Implementation as in code (design) is something different.

answered Jan 21 '18 at 19:07

MikeSW
14.6k ● 2 ● 30 ● 48