SOFTWARE ENGINEERING

Should the Data Model be identical to the domain model for mapping purposes?

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

2

2

Say I have a data model, which looks like this:

```
public class DataCustomer
{
    public virtual System.DateTime CreatedTime { get; set; }
    public virtual Guid Id { get; set; }
    public virtual string FirstName { get; set; }
    public virtual string Surname { get; set; }
    public virtual string FaxNumber{ get; set; }
    public virtual System.DateTime DateOfBirth { get; set; }
    public virtual String Gender { get; set; }
    public virtual string UserID { get; set; }
    public virtual List<Offers> Offers { get; set; }
}
```

This class is mapped to NHibernate. Now say I have a Domain Model like this:

```
    private virtual String Gender { get; set; }
    private virtual DateTime DateOfBirth { get; set; }
    public virtual List<Offers> Offers { get; set; }

    public DomainCustomer(Guid id, string gender, DateTime dateOfBirth)
    {
          Id=id;
          Gender=gender;
          DateOfBirth=dateOfBirth;
    }

    public void AssignOffers(IOfferCalculator offerCalculator, IList<Offers> offers)
    {
          //Assign the offers here
    }

}
```

Notice that the Data Model looks different to the Domain Model. I believe this is normal practice, however every example I look at online seems to show the Data Model being the same as the domain model.

Option 1

The advantage of them being identical is that mapping between the Domain Model and Data Model is very simple i.e. you can do this with AutoMapper:

```
DataCustomer dataCustomer = AutoMapper.Mapper.Map<DataCustomer>(domainCustomer);
```

Option 2

The advantage of them being different is that there is less data passed between the data model and domain model and vice versa. Also I think that it makes it slightly clearer i.e. a user/reader of my class knows what fields are needed by the domain model. With this approach; I would only map the members that have changed i.e. offers:

```
customerData.Offers = AutoMapper.Mapper.Map<string>(customerDomain.Offers);
```

Decision

Both options use a factory when mapping between CustomerData and CustomerDomain. The question is about the mapping between c# domain-driven-design

edited Feb 9 '18 at 10:12

asked Feb 9 '18 at 9:54
w0051977
6,173 5 35 71

Database modelling is for **minimizing data duplication**. Domain business modelling is 1) for designing your classes to reflect the business domain 2) and good class design is for **minimizing behavior duplication**. A business domain which looks like databse means either your db is not normalized or your business domain is not really a business domain. – CodingYoshi Feb 11 '18 at 16:42

It's really hard to discuss without a proper domain model. You are exposing the Offers so that any class can 1. replace/remove the list, 2. modify the list arbitrarily. Why is Offers exposed? – JimmyJames Feb 20 '18 at 17:37

@JimmyJames, do you know of a domain model (perhaps on github) or even a code sample where the ORM is not mapped directly to the domain model? I have resolved the issue with the Offers exposed. – w0051977 Feb 20 '18 at 17:47

4 Answers

4

For the sake of simplicity, let us first assume your "data model" reflects the underlying persistence model, probably a relational model behind it. Then it depends heavily on the size and structure of the overall system how much the data model will be different from the domain model:

- in small systems, there will be often only one core application. One starts with creating a small domain model, and the data model will be derived from it, solely for the purpose of persisting the domain model. In such a system, it is unlikely to have difference between the models. In fact, generating one from the other, or boiler plate code for all of them from a common meta source makes sense.
- bigger systems may be structured in a way where a common database in the background offers a larger data model shared by different applications or "bounded contexts", each accessing a different part of the system, maybe from a different point of view, but not completely disjoint. Normalization rules and the goal of storing each piece of information in one and only one place then can lead to differences between the data model and a domain model.

Note this is still a simplification, there are many more scenarios possible, like bigger systems using a microservice architecture, where each service is small and has its own persistence model, or bigger systems with a huge, shared domain model layer, where the domain model and data model are guaranteed to correspond to each other by generators. Other scenarios may have (1) a domain

So after this length introduction, what am I trying to tell? Well, my point is: if the data model and the domain model should be identical **is nothing which is dictated by the mapping strategy**.

It is quite the other way round: the overall architecture of the system tells you how the models are structured, and then you pick a suitable mapping strategy. Of course, where you are free to make choices in design, you should avoid to introduce unnecessary differences between the models as long as there is no real reason for doing so.

For example, names and types for corresponding attributes should be identical or follow strict mapping rules. That can lead to a situation where mapping all attributes of a class leads to simpler code than mapping only the subset of the attributes your program needs today. For this case, I would recommend to start with all attributes and optimize in case you notice this to be a real, measurable bottleneck.

edited Feb 9 '18 at 14:55

answered Feb 9 '18 at 13:12

Doc Brown
154k 27 289 444

Thanks. In my example; the Domain Model requires 2 properties in order to set the Offer List. Say the Data Model had 20 members - surely it would be a waste to also pass the 18 members that are not needed? - even in the early stages of the project? – w0051977 Feb 9 '18 at 16:39

@w0051977: you are overthinking this, and by constructing artificial examples you can prove anything you want. – Doc Brown Feb 9 '18 at 21:44

1 ... If you are working within a system where you have code generators at hand to generate all these 20 attributes and it takes more manual effort to restrict the code to just 2 of them than generating all 20, I would probably start with the 20 attributes. If you however, don't have such code generators, and the effort for writing the code for only 2 is smaller, I would take that route. But in a real system, there will be additional information about the architecture, one has to take that into account. – Doc Brown Feb 9 '18 at 21:54

So in short: there is no braindead, hard-and-fast rule for this, use your common sense and your experience. – Doc Brown Feb 9 '18 at 21:56

I agree with the "artificial examples" thing and I also agree that everything is not always black and white - there are always shades of grey. I have asked another question here though if you are interested: softwareengineering.stackexchange.com/questions/365643/…. It is more technical than theoretical so I have put it on Stackoverflow rather than Software Engineering. – w0051977 Feb 10 '18 at 10:15

6

First of all, having your domain model being different from your persistence data is normal. If you take the traditional hexagonal schema, you can see that persistence is reached through an "adapter".

Then, picking in between your options. I believe the mapping in between both should be within a (persistence-oriented) repository, rather than a factory. It sounds like a small naming difference, but it helps to make clear that a repository can do much more: for instance, when saving your `DataCustomer`, you may want to cascade save to several other data models

Finally, that repository can call Automapper or use any other mechanism. What is really important in the end is that built aggregates have their *invariants* respected (all required fields are filled in...). From that point of point, I don't quite see the difference between the 2 approaches.

In a nutshell: you should be concerned about the quality of your domain model, not about the technical means to build it. The domain layer is more important than the infrastructure layer.

edited Feb 9 '18 at 11:16

answered Feb 9 '18 at 10:30
romaricdrigon
161 4

1 Can you provide some code showing how the domain object is returned by the repository? (remember that the ORM maps to a data object). Do you use a method inside the repository to do the mapping? – w0051977 Feb 9 '18 at 21:41

What matters is not quite the code inside your repository methods, but its interface. The usual definition of the repository pattern is that it is an object *behaving like an in-memory collection* of instances: for example, your `DomainCustomerRepository` will have `get` method (by identifier), `find` (by a more complex specification), `add`, `remove` ... Inside, you will call NHibernate persistence methods + Automapper or similar, as they show in the Getting Started guide – romaricdrigon Feb 12 '18 at 16:28

1

A domain model and the data model have a different view of the world and should be expressed in the code.

The domain model should be modeled so that it aligns to the **business** view of the application while the data model is all about manipulating the domain model so that it's easy to be consumed by others for specific view/action related reasons (e.g. REST API, Web Page or Forms app...)

answered Feb 20 '18 at 17:18
Yitzchok
113 4

Can you answer my other question here: stackoverflow.com/questions/48884482/… ? – w0051977 Feb 20 '18 at 17:23

There is no reason why you **must** keep them the same, though it's *desirable*.

Imaging you are storing the info in an array of strings. Your data model would differ completely from your domain model.

Imaging your data model isn't yours (you are using a third-party API). Your business logic might differ from the business logic of that third-party.

The rule of thumb here is: you have to keep your domain models as close to your needs as you can.

answered Feb 11 '18 at 11:42

A Bravo Dev
234 1 4