How to avoid Anemic Domain Models and maintain Separation of Concerns?

Asked 16 years, 2 months ago Modified 13 years, 10 months ago

Viewed 4k times

20

It seems that the decision to make your objects fully cognizant of their roles within the system, and still avoid having too many dependencies within the domain model on the database, and service layers?

For example: Say that I've got an entity with a revision history, and several "lookup tables" that the data references, your entity object should have methods to get the details from some of the lookup tables, whether by providing access to the lookup table rows, or by delegating methods down to them, but in order to do so it depends on the database layer to read the data from those rows. Also, when the entity is saved, It needs to know not only how to save itself, but also to save entries into the revision history. Is it necessary to pass references to dozens of different data layer objects and service objects to the model object? This seems like it makes the logic far more complex to understand than just passing back and forth thin models to service layer objects, but I've heard many "wise men" recommending this sort of structure.

language-agnostic design-patterns oop

Share

Improve this question

Follow

asked Oct 22, 2008 at 23:01

Adam Ness

6,273 ● 4 ● 29 ● 39

4 Answers

Sorted by: Highest score (default)

19

Really really good question. I have spent quite a bit of time thinking about such topics.

You demonstrate great insight by noting the tension between an expressive domain model and separation of concerns. This is much like the tension in the question I asked about [Tell Don't Ask and Single Responsibility Principle](#).

Here is my view on the topic.

A domain model is anemic because it contains no domain logic. Other objects get and set data using an anemic domain object. What you describe doesn't sound like domain logic to me. It might be, but generally, look-up tables and other technical language is most likely terms that mean something to us but not necessarily anything to the customers. If this is incorrect, please clarify.

Anyway, the construction and persistence of domain objects shouldn't be contained in the domain objects themselves because that isn't domain logic.

So to answer the question, no, you shouldn't inject a whole bunch of non-domain objects/concepts like lookup tables and other infrastructure details. This is a leak of one concern into another. The Factory and Repository patterns from Domain-Driven Design are best suited to keep these concerns apart from the domain model itself.

But note that if you don't have any domain logic, then you will end up with anemic domain objects, i.e. bags of brainless getters and setters, which is how some shops claim to do SOA / service layers.

So how do you get the best of both worlds? How do you focus your domain objects only domain logic, while keeping UI, construction, persistence, etc. out of the way? I recommend you use a technique like Double Dispatch, or some form of restricted method access.

Here's an example of Double Dispatch. Say you have this line of code:

```
entity.saveIn(repository);
```

In your question, saveIn() would have all sorts of knowledge about the data layer. Using Double Dispatch, saveIn() does this:

```
repository.saveEntity(this.foo, this.bar,
this.baz);
```

And the saveEntity() method of the repository has all of the knowledge of how to save in the data layer, as it should.

In addition to this setup, you could have:

```
repository.save(entity);
```

which just calls

```
entity.saveIn(this);
```

I re-read this and I notice that the entity is still thin because it is simply dispatching its persistence to the repository. But in this case, the entity is supposed to be thin because you didn't describe any other domain logic. In this situation, you could say "screw Double Dispatch, give me accessors."

And yeah, you could, but IMO it exposes too much of how your entity is implemented, and those accessors are distractions from domain logic. I think the only class that should have gets and sets is a class whose name ends in "Accessor".

I'll wrap this up soon. Personally, I don't write my entities with saveIn() methods, because I think even just having a saveIn() method tends to litter the domain object with distractions. I use either the friend class pattern, package-private access, or possibly the Builder pattern.

OK, I'm done. As I said, I've obsessed on this topic quite a bit.

Share Improve this answer Follow

edited May 23, 2017 at 12:25
Community Bot
1 ● 1

answered Oct 23, 2008 at 0:16
moffdub
5,314 ● 2 ● 37 ● 32

2 In short, you're proposing accessing the repository from the domain? – aaimnr Dec 10, 2009 at 12:26

1

"thin models to service layer objects" is what you do when you really want to write the service layer.

ORM is what you do when you don't want to write the service layer.

When you work with an ORM, you are still aware of the fact that navigation *may* involve a query, but you don't dwell on it.

Lookup tables can be a relational crutch that gets used when there isn't a very complete object model. Instead of things referencing things, you have codes, which must be looked up. In many cases, the codes devolve to little more than a static pool of strings with database keys. And the relevant methods wind up in odd places in the software.

However, if there is a more complete object model, we have first-class *things* instead of these degenerate lookup values.

For example, I've got some business transactions which have one of *n* different "rate plans" -- a kind of pricing model. Right now, the legacy relational database has the rate plan as a lookup table with a code, some pricing numbers, and (sometimes) a description.

[Everyone knows the codes -- the codes are sacred. No one is sure what the proper descriptions should be. But they know the codes.]

But really, a "rate plan" is an object that is associated with a contract; the rate plan has the method that computes the final price. When an app asks the contract for a price, the contract delegates some of the pricing work to the associated rate plan object.

There may have been some database query going on to lookup the rate plan when producing a contract price, but that's incidental to the delegation of responsibility between the two classes.

Share Improve this answer Follow

answered Oct 22, 2008 at 23:40

S.Lott
391k ● 82 ● 517 ● 788

I aggree with DeadBeef - therein lies the tension. I don't really see though how a domain model is 'anemic' simply

1

because it doesn't save itself.

There has to be much more to it. ie. It's anemic because the service is doing all the business rules and not the domain entity.

```
Service(IRepository) injected

Save(){

DomainEntity.DoSomething();
Repository.Save(DomainEntity);

}

'Do Something' is the business logic of the domain
entity.

**This would be anemic**:
Service(IRepository) injected

Save(){

if(DomainEntity.IsSomething)
  DomainEntity.SetItProperty();
Repository.Save(DomainEntity);

}
```

See the inherit difference ? I do :)

Share Improve this answer

Follow

edited Feb 5, 2011 at 0:41

Chandu

82.8k ● 19 ● 135 ● 135

answered Dec 11, 2009 at 19:12

Steve

387 ● 4 ● 12

0

Try the "repository pattern" and "Domain driven design". DDD suggests to define certain entities as Aggregate-roots of other objects. Each Aggregate is encapsulated. The entities are "persistence ignorant". All the persistence-related code is put in a repository object which manages Data-access for the entity. This way you don't have to mix persistence-related code with your business logic. If you are interested in DDD, check out eric evans book.

Share Improve this answer

Follow

answered Oct 22, 2008 at 23:08

Manu

29.1k ● 28 ● 77 ● 83

2 It doesn't answer the question. The question is "Should the repository be injected into domain"? If so, it would breaks Separation of Concerns. On the other hand, if those actions would be performed by service, it could lead to Anemic Domain Model. – aaimnr Dec 10, 2009 at 12:19