Anaemic Domain Model vs. Rich Domain Model

BLOG | By Ismael Mota | 9th October 2018

Some time ago I found myself arguing with a developer about the approach to take when it came to some design considerations of a microservice solution we were working on. I remember talking about some principles that we needed to consider and trying to inspire my counterpart to adopt a more DDD approach. Although what I was saying made sense in his head, he answered with "*this is not a DDD project, this is a microservice project!*"

What is an Anaemic Domain Model

What we were facing were the problems generated by an Anaemic Domain Model. An Anaemic Domain Model is a model with no logic in it. Domain classes look more like a bunch of public setters and getters without domain logic where the client of the class has control over how to instantiate and modify the class. In these models, the client has to interpret the class purpose and use. Usually, the logic has been pushed to other classes called something like *services*, *helper* or *manager* plus the name of the domain class. With the logic sitting in another class, there is nothing that helps the client to navigate or use the model class.

This approach leads to many design problems and disadvantages:

- Encapsulation is violated
- Hard to maintain
- Duplicated business logic
- Cannot assure the entities in the model are in a consistent state
- Low cohesion
- Favours gap and misunderstanding between development and business

Rich Domain Model

A good way to address the main disadvantages of Anaemic Domain Models is implementing a Rich Domain Model. The main difference with an Anaemic Domain Model is that our domain logic is part of our domain entities, data and behaviour sit together. That logic guides and controls how the entity is instantiated, validated and operated, preventing the client from having entities with an inconsistent state.

Ways to avoid an Anaemic Domain Model

There is a long list of best practices you can use to avoid an Anaemic Domain Model, also it depends on the complexity of the solution and on how purist you want to go when comes to enriching your model. The objective of this post is not to

AMIDO

mplete study or exhaustive list of best practices, but here are a few of hints and tips to get you started:

- Define Invariants. Invariants are what make your entities to be an entity. It is their essence and motive for them to exist. It is what makes an entity Order being an order. Can an order exist without an order date?
- Think twice before using public parameterless constructors. It is very possible, like in the previous example, that in order to have a valid state your objects need some initialisation data. Prevent your clients from creating your objects in an inconsistent state.
- Get rid of public property setters for similar reasons. Don't let the client decide on the internal representation of the entity; create methods that operate on these properties instead.
- Push your behaviour and domain rules from domain services to your domain model. If you find yourself creating a domain service, think again about your design and your entities. If you still feel that you need to create a domain service, try to use decorators instead. If nothing works, then create that domain service, but make sure you define it in terms of the language of the model and make it stateless. Bear in mind that domain services and domain behaviour are not the same concepts. Domain services act more as entities intermediaries while domain entities carry the domain behaviour.

In summary, embrace encapsulation! It is always a good practice to assign the responsibility of maintaining data integrity to the objects that contain that data.

So, why are developers still using Anaemic Domain Models?

Martin Fowler wrote an article in 2003 where he classified the Anaemic Domain Model as an anti-pattern. Since then, the Anaemic Domain Model has been widely recognised as an anti-pattern and even though it is not uncommon to find many projects using it, especially in combination with the three-layer architecture. I think that there are a couple of reasons for this:

- Active Record pattern. "The active record pattern is an approach to accessing data in a database. A database table or view is wrapped into a class. Thus, an object instance is tied to a single row in the table". It is very popular and the domain objects are generated automatically in combination with an ORM tool. This produces Anaemic Domain Model objects that are just bags of public setters and getters.
- Straightforward use. Procedural programming is easier than Object Oriented Programming. Anaemic Domain Models have the domain logic in some other service classes. When you pull your domain behaviour to domain services classes you are basically taking an object, operating upon it and its data, and returning it to the client. Bad news, that is not OOP, it is procedural programming.
- Solutions with low logic requirements. For some solutions with no domain logic associated with mainly CRUD operations, an Anaemic Domain Model using an MVC architecture could be enough.
- Some articles are trying to challenge the anti-pattern classification of Anaemic Domain Models. For instance, this article argues that Anaemic Domain Models adhere better than Rich Domain Models to the SOLID principles, especially when it comes to the Single Responsibility Principle. I would invite you to spot the issues behind the reasoning in the article though.

Conclusion

Since I read Domain-Driven Design by Eric Evans a long time ago, I always try to follow and apply some of the principles and patterns described in the book to my designs. However, from time to time I wonder if I put excessive emphasis and effort on it .

Sometimes the complexity of the project simply doesn't invite to apply many of the principles, sometimes we can deliver and comply with our requirements with less complexity, and sometimes you simply YAGNI. If your solution mainly contains CRUD operations without business logic embedded, you are unlikely to need DDD.

Going back to "*this is not a DDD project, this is a microservice project!*". We were already facing many of the drawbacks of having an Anaemic Domain Model. Also, many concepts and patterns described in Domain-Driven Design from Eric Evans microservices architectures, even when those principles were described way before microservices were s the architecture that nowadays is being used in the industry. A couple of good examples of these princip are the use of the notion of Bounded Context to help determine the domain of a microservice and making your code speak the ubiquitous language. The ubiquitous language is going to ease communication and give a stronger sense of purpose to your domain design. Developers should not be unaware of business language.

Looking to develop new services alongside existing technology?

Let us help you take the technology you already have, and make it fit for the future.

Get in touch

Phone: 0203 176 4690

Email: enquiries@amido.com

Your Email Address

Home

Blog

News & Events

What we do

Careers

Contact

Terms and conditions