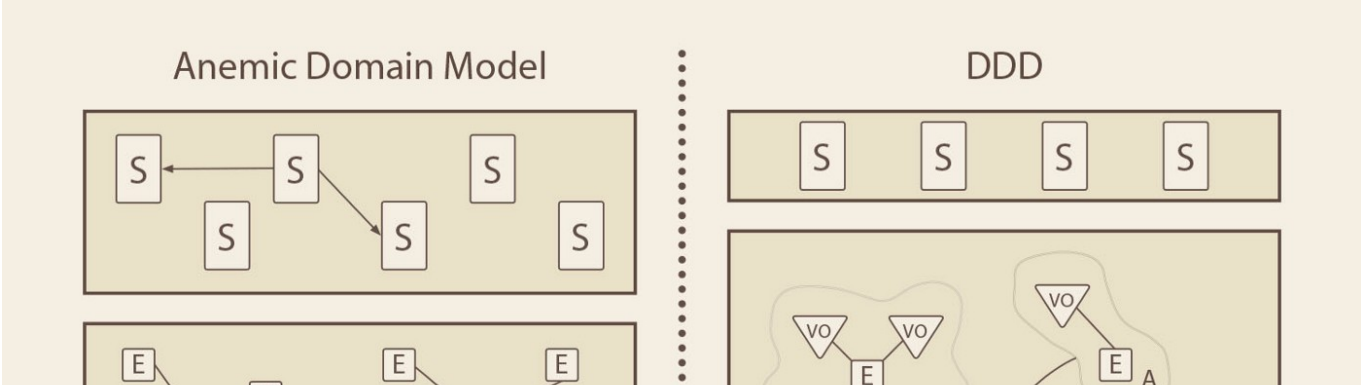
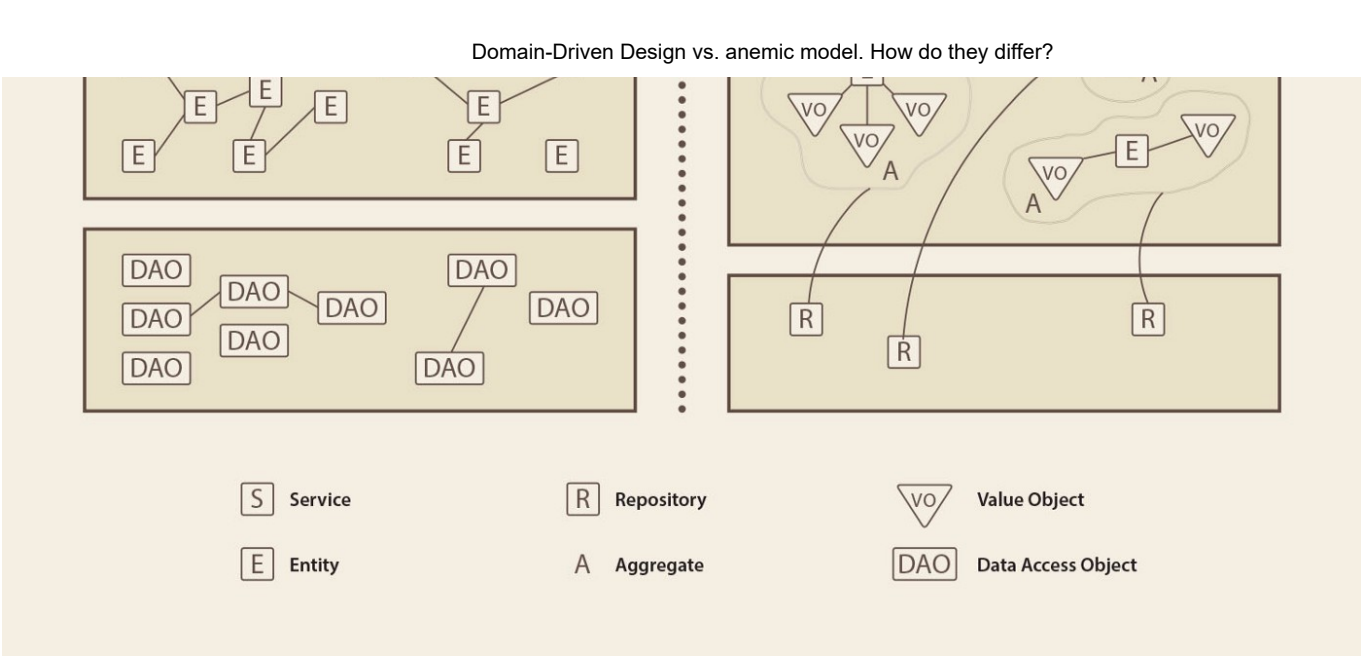
Domain-Driven Design vs. anemic model. How do they differ?

Kamil Berdychowski Follow

Jan 3, 2017 · 4 min read

Some time ago during one of our internal DDD learning group meeting I was wondering what is the best way to show differences between writing an application with an anemic model and application with applied concepts of DDD. I came up with a drawing which illustrates these two types of application.





Comparison of a common layered application and application with DDD architecture.

Anemic model and bulky services

Anemic domain model is nothing more but entities represented by classes containing only data and connections to other entities. These classes lack of the business logic, which usually is placed in services, utils, helpers etc. This design (three layers shown on the left side of the picture) is the natural way we model classes responsible for business cases.

When we implement a specific functionality, we first talk about the classes which will be persisted. We call them entities. They are represented as the

letter E in the graph on the picture above. These entities are in fact object-oriented representation for the database tables. In result, we don't implement any business logic inside them. Their only role is to be mapped by some ORM to their database equivalents.

When our mapped entities are ready, next step is creating classes for reading and writing them. We end up with the DAO (Data Access Objects) layer. Usually, each of our entities represents separate business case, so the number of DAO classes match the number of entities. They do not contain any business logic. DAO classes are nothing more than tools for retrieving and persisting entities. We can use an existing framework to create them, ie. spring-data with hibernate underneath.

The last layer, above the DAO, is the quintessence of our implementation — Services. Services make use of DAO in order to implement the business logic for specific functionality. Regardless of the number of functionalities, a typical service always performs following operations: loads entities using DAO, modifies their state according to requirements and persists them. Such architecture has been described by Martin Fowler as a series of Transaction Scripts. The more complicated the functionality, the bigger number of operations in between loading and persisting. Often some services start to make use of the other services, resulting in growing in tons

of code, but only in the services. Entities or DAO stay the same, unless we provide new fields for persisting.

DDD comes with totally different approach in terms of placing the code in layers.

Rich model and thin services

Common architecture with Domain Driven Design model is presented on the right side of the picture.

Take note of a layer of services which is much thinner than his equivalent in an anemic model. The reason is that the most of business logic is included in Aggregates, Entities and Value Objects. In the services we put only operations working on many areas of the domain and logically fitting to none of these.

The domain layer has more types of objects. There are the Value Objects , the Entities and objects which assemblies these — the Aggregates. Aggregates can be connected only by identifiers. They cannot share any other data with each other.

The last layer is also thinner than in the previous model. Number of repositories corresponds to the number of aggregates in DDD so the application with the anemic model has more repositories than the one with the rich model.

Easier changes and less bugs?

Applying Domain Driven Design architecture gives developers some important benefits.

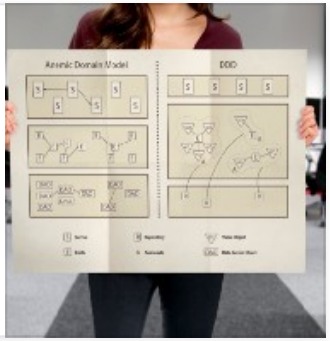
Thanks to dividing objects on the Entities and the Value Objects we can more precisely managing newly created objects in our application. Aggregates allow us to encapsulate parts of our domain so our API becomes simpler and changes inside aggregates are easier to implement (we don't need to analyse our changes impact on some other parts of the domain because it doesn't exist).

Less number of connections between domain elements reduce the risk of creating a bug while developing and modifying the code. Extended domain representation makes the code better reflecting how the domain experts describe the business rules. As a result, much easier is concluding about a correctness of solution, and if necessary, changing and developing it.

We encourage all of you to try DDD on daily basis. If you need some knowledge or support take a look at Eric Evans book Domain-Driven Design: Tackling Complexity in the Heart of Software or, maybe a little bit easier to understand, Implementing Domain-Driven Design written by Vaughn Vernon. We will try to give you a little bit of DDD concepts in next posts. Stay tuned.

Refactoring from anemic model to DDD

blog.pragmatists.com



Have you liked the post?

Sign up for our newsletter to get the newest posts! No spam, we promise!

Email

Sign up

Visit your domain objects to keep 'em legit!

Data correctness and enforcing business rules of given domain are one of those aspects, that are common to practically...

blog.pragmatists.com

Retrying consumer architecture in the Apache Kafka

blog.pragmatists.com

. . .

Find Pragmatists on Facebook and Twitter

Software Craftsmanship | Ddd | Backend | Java | Domain Driven Design