Domain Driven Design: A "hands on" Example (Part 2 of 3)

Suarte

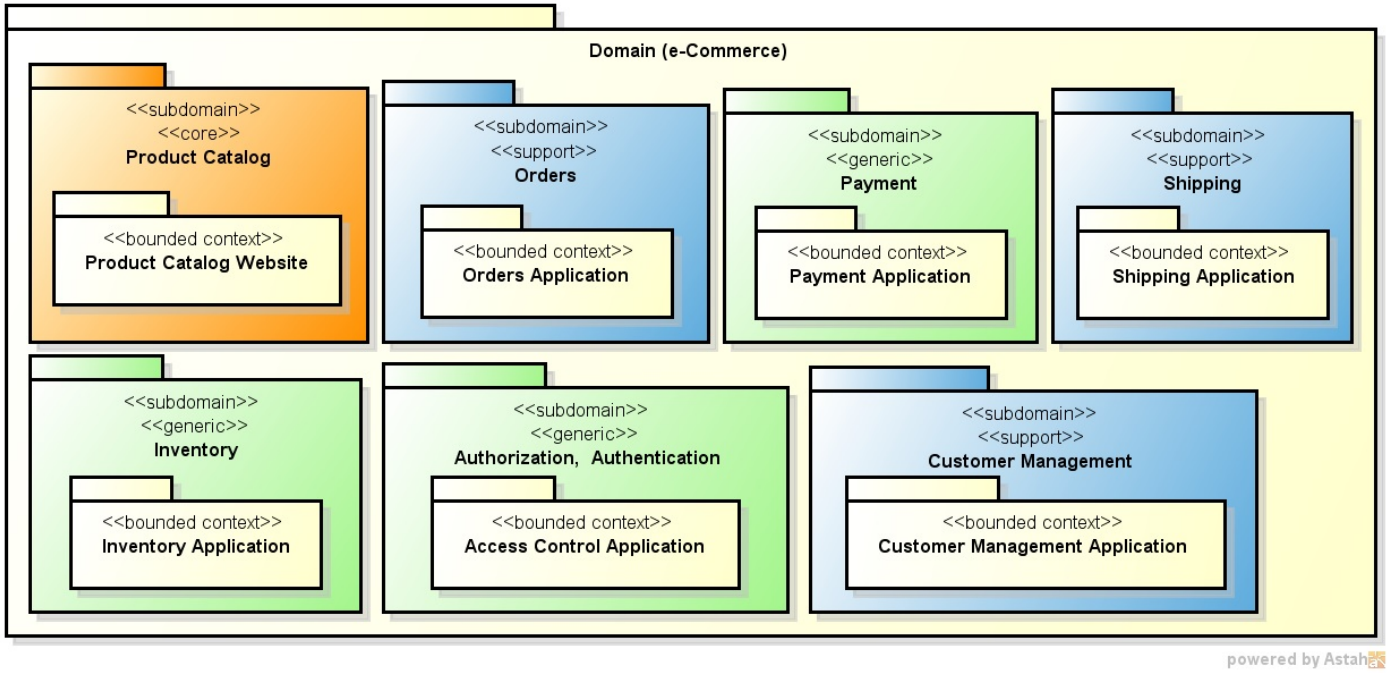
20 Apr 2016 CPOL

Rate this: ★★★★★ 5.00 (1 vote)

Hands on example of domain driven design - Part 2

CodeProject I am glad that this series introducing Domain Driven Design has pleased you readers. Thank you guys for all the positive feedback I have received. So, let's start the second part!

Just to remind you, I ended up with the following solution in the first post of this series:



Picture 1: A DDD approach to the e-Commerce Domain

I am going to pick only two Bounded Contexts as examples for modeling. As I said before, some of these contexts can even be supported by off-the-shelf applications, as the intent here is not to develop the whole solution basically from scratch. I thought it would be a good thing if I worked on the contexts of the "Product Catalog Website" (the Core Subdomain) and a Support Subdomain (my choice was the "Orders" one).

Concepts for Creating Domain Class Models in a DDD Fashion

First of all, I would like to state here that any OOD (Object Oriented Design) know-how is useful for constructing Domain Class Models, it doesn't matter if you are a DDD enthusiast or not. For example, I like to use concepts from, GRASP (General Responsibility Assignment Software patterns - Craig Larman) and GOF patterns when I am creating my models. A solid understanding of UML is very useful too. Let's take a look in some core concepts that you must know when modeling a Domain Class Model.

Entity

It is possible to find many definitions for "Entity" in literature. It can represent many different things. However, for DDD, the meaning of Entity is very clear. Vernon's book, "Implementing Domain Driven Design", has an excellent definition for Entity:

"We design a domain concept as an Entity when we care about its individuality, when distinguishing if from all objects in a system is a mandatory constraint. An Entity is a unique thing and is capable of being changed continuously over a long period of time."

Mutability and unique identity are the two main characteristics entities have.

Value Object

Your capacity to identify Domain concepts and model them as "`Value Objects`" is one of the most powerful tools you can use in order to succeed in creating DDD models. Why? Well, if you are not able to identify them, you tend to model everything as an "`Entity`".

It is hard to portray all the nuances of it in a post as short as this one. Let's try to make it clear in the class model (further). For now, keep in mind that "`Value Objects`" do not need to be treated as unique. Normally, they are immutable and their methods (when available) should provide a "Side-Effect-Free" behavior (not changing the object state).

Aggregate

Think in an Aggregate as a block composed for different pieces and, even that these pieces might exist by them self inside your context, it would not make any sense using them separately. Let's see this again in the class model section.

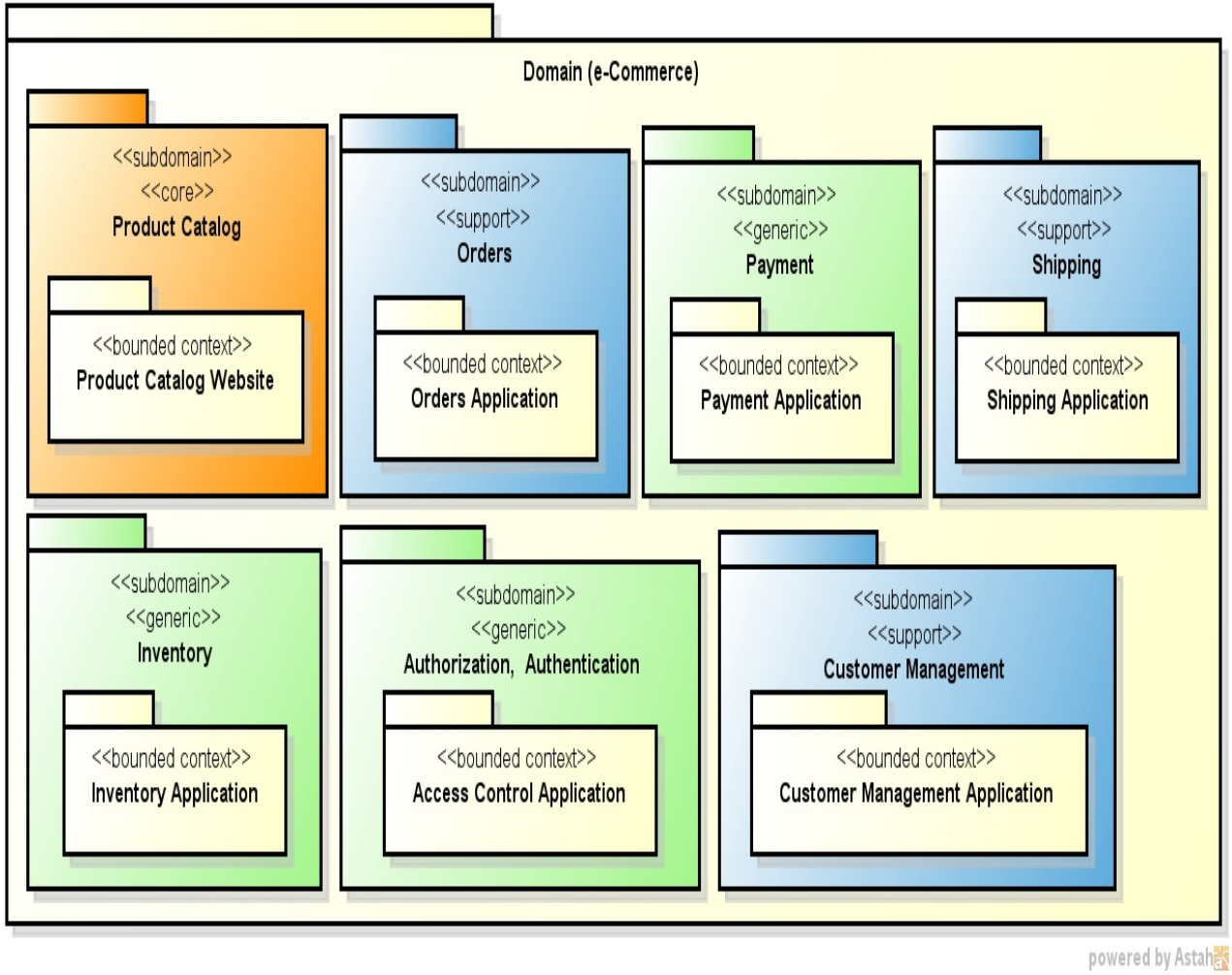
Modeling

Well... it is time to start modeling and I have one important tip: when modeling, FORGET about data models (relational models) and do not think about persistence! It can really harm the way you see your modeling scenario. If you want to succeed in applying DDD, this mental approach is necessary.

These are just hypothetical models which I created in order to support our example. Please do not expect fully functional models!

The "Product Catalog" Context Model

Based on everything that has been discussed above, this is my solution for the "Product Catalog" context:



Picture 2: The "Product Catalog" context model

Now, let's dig into the details of this simple class model. First of all, it is possible to notice that there is only one class modeled as "`Entity`", regardless of the fact this model is a small one. Why?

The Classes "`Weight`" and "`Dimension`" are very similar: both represent a value and how it should be interpreted. If you look at these classes' methods, you will notice that there are no "set" methods. An instance of "`Weight`", for instance, should have all of its attributes set when it is constructed. In order to achieve this, I might use a Constructor with all the parameters or a kind of "`Creator`" method.

Remember what was said about "`Value Object`" above? I don't need to change its data once it has been created. It works well being immutable and consequently allows me to be free of all the complexities related to "Entities".

In the case of the class "`Review`", well... I believe most people would model it as an "`Entity`". But I think it fits well as a "Value Object". Its relationship with "`Product`" does not have a strong meaning inside the context. A "`Product`" does not depend on "`Review`" to be available in the catalog. Once an instance of "`Review`" is created, I do not see any reason to change it. So, I decided to model it as a "`Value Object`" too.

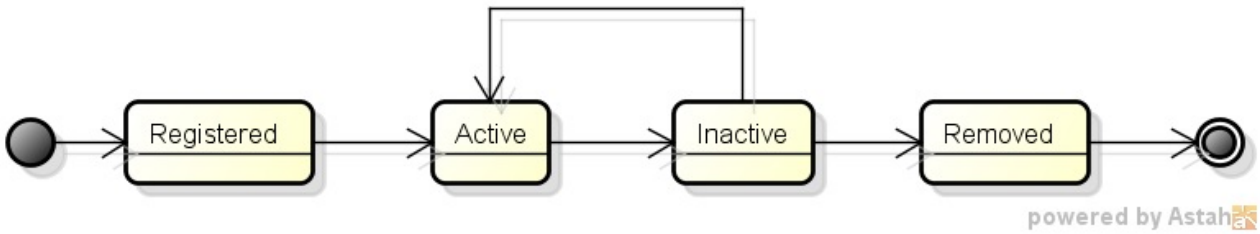
Finally, the "`Product`" class. It is an "`Entity`" for sure because I need it being uniquely identifiable. Also, I must be able to change its state and might even want to track changes on it. It is an "Aggregate Root" too because it aggregates both "`Shipping Weight`" and "`Dimension`" in a composition relationship.

The real meaning of aggregation here is: you should see those objects as inseparable. For example: once a instance of the class "`Weigh`" is created and set to a instance of "`Product`" as the "`Shipping Weight`", every time you load "`Product`" from your persistence layer, it should contains the "`Shipping Weight`" too. That is why you should be careful when modeling aggregates in order not to create monsters which will bring you problems (probably related to persistence complexity and performance).

Now the "`Product`" class methods. Can you explain why it is apparently missing some set methods? Why doesn't it have a "`get`" method to return the list / array of pictures? What the hell are those methods "`activate()`" and "`deactivate()`"?

In order to create an instance of "`Product`", you must supply at least the product ID and the inventory code (product identifier coming from another bounded context, the inventory context). This data is a kind of immutable data and it would not make any sense to allow a "`Product`" instance creation without it. Before you ask, I prefer not to use database features like identity columns, sequences and so on for ID generation. This way, I've got my "`Entity`" classes not dependent on persistence in order to get an unique identifier.

Moving on, I modeled "`Product`" having the following "`Status`" state machine:



Picture 3: The "Product" class "Status" state machine

This way, the system would support a product registration for future data input without resulting on it automatically appearing in the catalog.

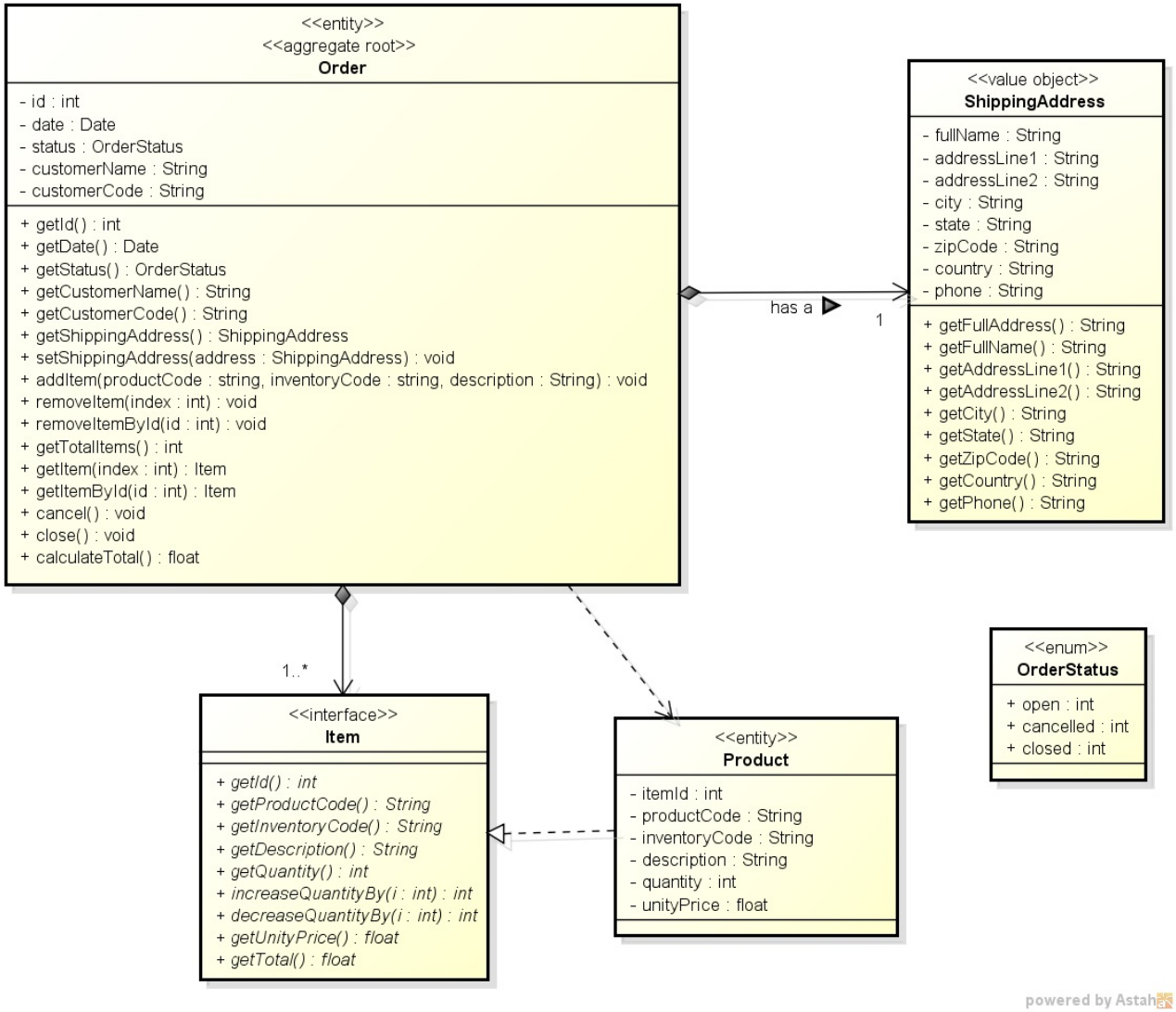
Without exposing the list / array of pictures, the "`Product`" class has total control over any business logic regarding product pictures. Maximum number of pictures, position, duplicated pictures, etc; all these kind of rules can be managed and ensured by the expert class. So, I've got more cohesion and, in some cases, low coupling too.

The method "`activate()`" is crucial: it must guarantee that, before changing the state of an instance to "`Active`", all required data have been provided and there is no violation regarding machine status flow. The "`deactive()`" method should play a similar role on treating its pertinent business rules.

In this manner, applying all of these concepts, you avoid what many authors call of "Anemic Domain" (Fowler, Martin) once your classes provide real business functionality and not only data.

The "Orders" Context Model

All concepts previously discussed are valid for the "`Orders`" context too. So let's take a look at the model:



Picture 4: The "Orders" context model

The "`Order`" class is an "`Entity`" and "`Aggregate`" by same reason explained above (in the "Product Catalog" model). It has methods to control all its pertinent business rules like, "`cancel()`", "`close()`", "`calculateTotal()`" and so on; it controls how new instances of "`Item`" are created, added and removed. Therefore, we can say that it is an expert and has high cohesion.

The "`Product`" class here represents a totally different concept. It basically represents an "`Order Item`". I modeled an interface "`Item`" just to add a little bit more of "low coupling" to the solution. Imagine that product data will come from other context ("`Product Catalog`" context), but the "`Order`" application logic should be unaware of this.

In this manner, application classes interacting with the "`Order`" class don't need even know about the existence of the "`Product`" class, even though it has a different meaning and its own context. I haven't talked about application logic classes before and I will explain it in the last part of this series.

Conclusion

I tried to be succinct and clear as much as possible in order to share how I usually think when I model using DDD concepts. I hope these tips and insights might be useful for you guys. I would love to be in touch by email, comments, etc. with anyone who wants to discuss about this subject. In the third and last part, I will show some code examples. Best regards!

Related Posts

- Domain Driven Design: what is it really about?
- Domain Driven Design: a "hands on" example (part 1 of 3)

License

Share

About the Author

Suarte

Technical Lead PSafe Technology

Brazil

A software development lover with more than 10 years of experience on programming, the IT field and computer science as well.

Comments and Discussions

You must Sign In to use this message board.

Search Comments

Spacing Relaxed ▼ Layout Normal ▼ Per page 25 ▼ Update

-- There are no messages in this forum --