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

Suarte

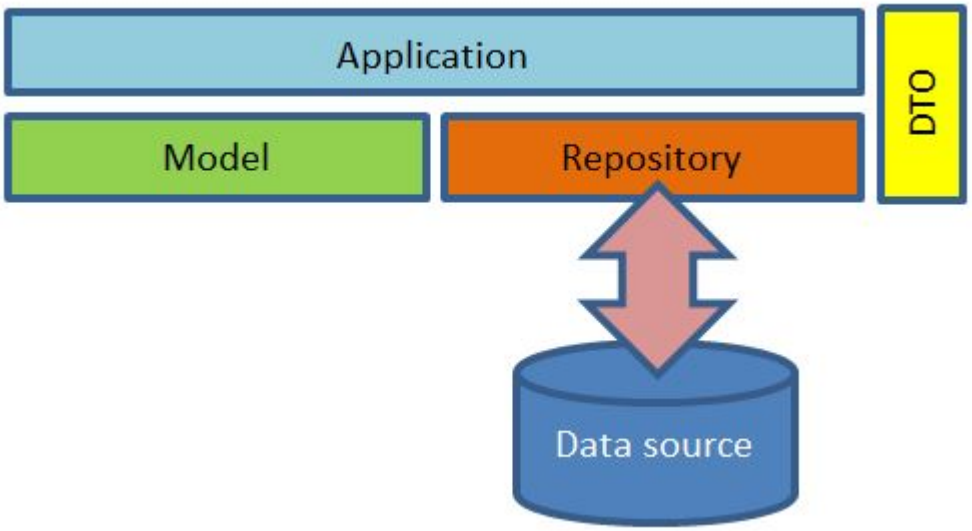
20 Apr 2016 CPOL

A hands on example of domain driven design - Part 3

CodeProject Finally, we get to the last part of this series. Let's dig into some code example for the "Product Catalog" context. The code showed here is just for tutorial purposes and should not be used in real production environment. If you decide to use it, you do so at your own risk. The target here is to expose how it is possible to translate some of the discussed concepts into code.

Logical Layering

First of all, I would like to introduce the logical layered architecture that I like to use in my applications. It is a kind of "relaxed layer", pretty simple, like this:



Picture 1: Application logical layered architecture

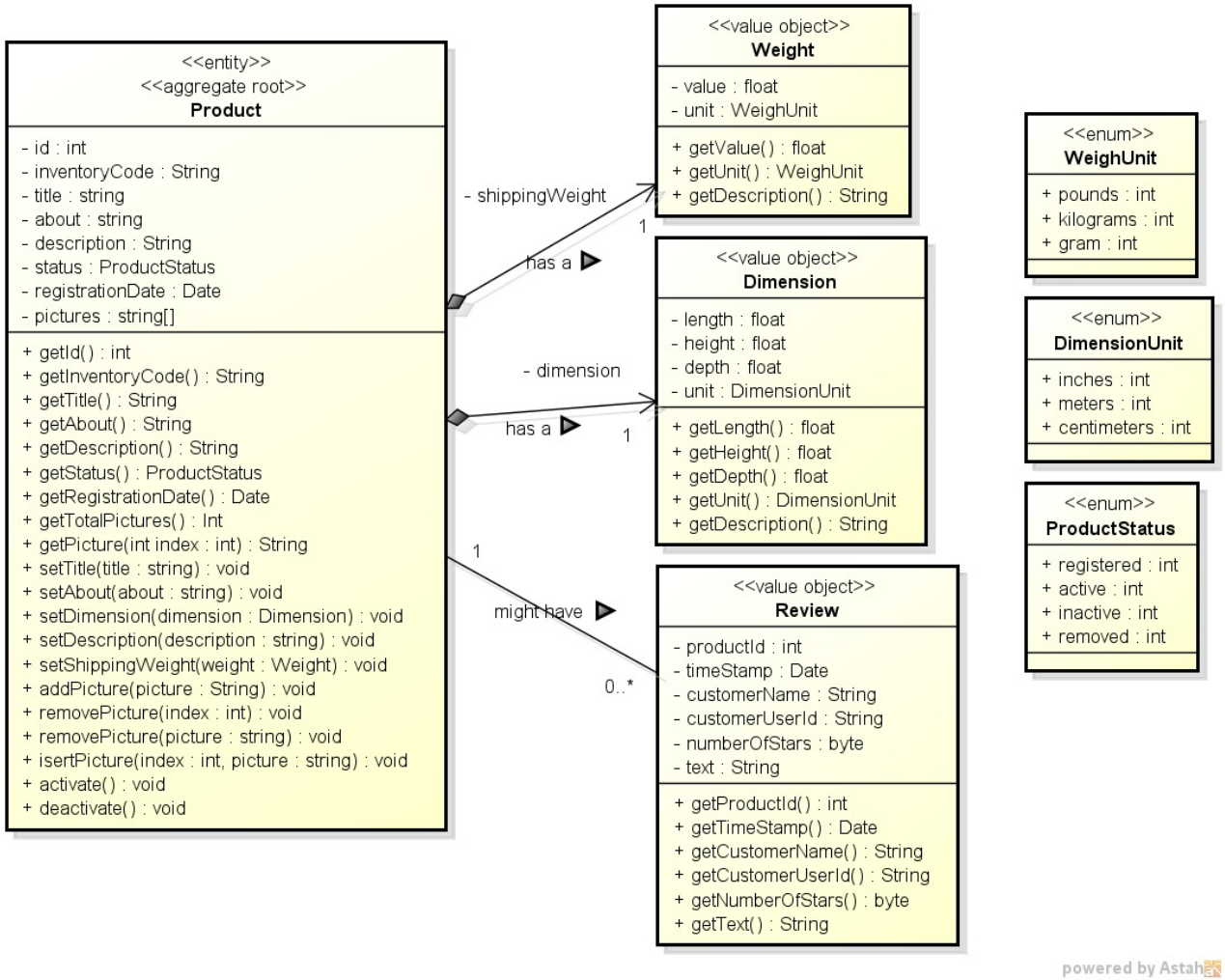
Notice that when I say "my applications", I mean the business layer, the part of the system that solves my business problem. Other layers like view, interface and utility layers are not considered.&

In this manner, the "Application" layer acts like a Facade to the model so that it is not exposed to other external layers. The "Model" layer is where the model reside itself, while the "Repository" layer is in charge of persistence, whatever the type of the data source has been used (database systems, files, etc.).

The DTO (Data Transfer Object - "Patterns of Enterprise Application Architecture" Fowler, Martin) layer, is the way my application exposes data. I like to use it so I don't have to expose my Domain objects and its interfaces, increasing the overall level of low coupling in my system.

Code View: The Model Layer

This is the Model for the "`Product Catalog`" context from the previous post:



Picture 2: "Product Catalog" context model

I tend to write my classes pretty close to what I modeled. Why? I think it is the gist of applying Domain Driven Design: write your code as close as possible to your model and, therefore, your business. You always end up adding or modifying something when you are coding and materializing your abstract model. Totally normal.

Let's take a look at specific parts of the "`Product`" class and its implementation. The first one, the constructor:

```
public Product(Guid id, string inventoryCode, string title)
{
    if (id == null)
        throw new Exception("Invalid Id!");

    if (string.IsNullOrWhiteSpace(inventoryCode))
        throw new Exception("Invalid Inventory Code!");

    this.id = id;
    this.inventoryCode = inventoryCode;

    this.ChangeTitle(title);

    this.registration = DateTime.Now;
    this.status = ProductStatus.Registered;

    this.pictures = new List<string>();
}
```

Picture 3: "Product" class "Constructor" method

What is important to notice here is that this constructor guarantees the least amount of information necessary for creating a new "`Product`" instance. In my model, it would not make any sense to have an instance of "`Product`" without an "`id`", an "`inventory code`" and a "`title`". Provide me at least these and I will give you an instance of "`Product`" in the status "`Registered`", so you can work on it further. That is the idea.

Now, let's take a look at some methods:

```
public void ChangeTitle(string t)
{
    if (string.IsNullOrWhiteSpace(t))
        throw new Exception("Invalid title! Please, provide a valid one.");

    this.title = t;
}
```

Picture 4: "Product" class "ChangeTitle" method

"`Title`" is a mandatory data. There is no way to create an instance of "`Product`" without providing it. On the other hand, it is possible to change and modify it. Can you figure out the semantics difference in the method name? The method could be simply named as "`SetTitle`" (a very common approach) indeed.

However, it is pretty much intuitive to call it "`ChangeTitle`", once it cannot be really set. As it is needed when you created the instance, it would not make sense to set it. It might seem not important at all, but this subtle naming approach is what makes your implementation closer to your real business rule in this case.

Furthermore, the method assures that only valid changes take place. Any other kind of business validation regarding "`Title`" (like maximum size allowed, not allowed words, etc.) would fit well in this method.

Let's see another one:

```
public void Activate()
{
    //guarantees the "machine state" logic
    if (this.status == ProductStatus.Registered || this.status == ProductStatus.Inactive)
    {
        //verifies if all the necessary data have been set. A product cannot be activated without Dimension, for example

        if (string.IsNullOrWhiteSpace(this.description) || string.IsNullOrWhiteSpace(this.about))
            throw new Exception("Product cannot be activated! Please, make sure to provide product details like 'description' and 'about'");

        if (this.dimension == null || this.weight == null)
            throw new Exception("Product cannot be activated! Please, make sure to provide product measures like 'dimension' and 'weight'");

        if (this.listPrice == 0)
            throw new Exception("Product cannot be activated! Please, make sure to provide the 'list price' for this product");

        //and if everthing is ok... change the status!
        this.status = ProductStatus.Active;
    }
    else
        throw new Exception(string.Format("Cannot activate this product! Current status is '{0}'", this.status));
}
```

Picture 5: "Product" class "Activate" method

The "`Activate`" method is interesting. A product should be only activated if it attends to several business requirements. These requirements are tested in the method. If all of them are good, the status of the instance is changed from "`Registered`" to "`Active`". This is your business logic being handled by your model instead by other layers of your application.

Code View: The Application Layer

As I said before, the Application layer works like a Facade to the Model, so that I can avoid high coupling of my Model to other layers. The Application layer usually is used to support business transaction rules. It is used to coordinate operations executed on the Model and can provide general application support as well, like authentication, security, integration and so on.

I like to see my Application classes as a kind of "Transaction Script" - "Patterns of Enterprise Application Architecture" Fowler, Martin - coordinating model classes in order to achieve some business transaction, as explained above.

As an example, I created a "`CatalogService`" Application class in order to support business transactions regarding the "`Product Catalog`" context:

```
namespace com.thecoderlife.example.productcatalog.application
{
    //Represents a "Catalog" Business Service
    public class CatalogService
    {
        /// <summary> ...
        public string RegisterNewProduct(string inventoryCode, string title)...

        /// <summary> ...
        public void ProvideProductDetail(string productId, string description, string about)...

        /// <summary> ...
        public void AddPicture(string productId, string filePath)...

        /// <summary> ...
        public void ProvideDimensionAndWeight(string productId, float lenght, float height, float depth, string dUnit, float weight, string wUnit)...

        /// <summary> ...
        public void ProvideProductListPrice(string productId, float price)...

        /// <summary> ...
        public void ActivateProduct(string productId)...

        /// <summary> ...
        public IList<CatalogItem> GetCatalogItems()...
    }
}
```

Picture 6: "CatalogService" Application Class

There are three important things that should be noticed in this class. The first one, it does not expose the Model. It works only with native data and DTO interfaces (`CatalogItem` is a DTO).

Second, it provides only real business transactions methods, all pertinent to the "`Product Catalog`" context. There is no such generic method like "`UpdateProduct`", which would transfer any business transaction responsibility to its caller.

Third and last, it can be used in many different situations in my system. I can build any graphic interface layer and couple it to this layer. I can build a service layer (Rest, SOAP, etc.) and put it in front of this layer (a way to expose business logic remotely). I can build integration with other contexts through the Application layer.& All of these aspects contribute in order to have an easily maintainable and robust business application.

Let's take a look at some of these methods in detail:

```
/// <summary>
/// Provides information about dimension and weight
/// </summary>
/// <param name="dUnit">Unit of measure for "Dimension"</param>
/// <param name="wUnit">Unit of measure for "Weight"</param>
public void ProvideDimensionAndWeight(string productId, float lenght, float height, float depth, string dUnit, float weight, string wUnit)
{
    Guid id = Guid.Parse(productId);

    ProductRepository rep = new ProductRepository();
    Product product = rep.Get(id);

    DimensionUnit dimensionUnit = (DimensionUnit) Enum.Parse(typeof(DimensionUnit), dUnit);
    WeightUnit weightUnit = (WeightUnit)Enum.Parse(typeof(WeightUnit), wUnit);

    Dimension dimensionObj = new Dimension(lenght, height, depth, dimensionUnit);
    Weight weightObj = new Weight(weight, weightUnit);

    product.SetDimension(dimensionObj);
    product.SetWeight(weightObj);

    rep.Persist(id, product);
}
```

Picture 7: "CatalogService" class "ProvideDimensionAndWeight" method

```
/// <summary>
/// Activate a product (make it avaiable in the catalog)
/// </summary>
public void ActivateProduct(string productId)
{
    Guid id = Guid.Parse(productId);

    ProductRepository rep = new ProductRepository();
    Product product = rep.Get(id);

    product.Activate();

    rep.Persist(id, product);
}
```

Picture 8: "CatalogService" class "ActivateProduct" method

```
/// <summary>
/// Returns a list of CatalogItems (products which are active)
/// </summary>
/// <returns>IList CatalogItem</returns>
public IList<CatalogItem> GetCatalogItems()
{
    ProductRepository rep = new ProductRepository();

    IList<CatalogItem> items = rep.GetCataloItems();

    return items;
}
```

Picture 9: "CatalogService" class "GetCatalogItems" method

The first two exemplified methods (`ProvidDimensionAndWeight` and `ActivateProduct`) follow a very similar pattern: parse data, query the repository, coordinate on the model and persist object's state. The last one, "`GetCatalogItems`", has a different pattern: it basically queries the repository and returns data. Persistence techniques are out of the scope of this post. However, it is a good opportunity to talk about some DDD concepts regarding persistence.

"There is no silver bullet". Domain Object Models are good to organize and tackle business complexities. On the other hand, they might not be good for data retrieving in many scenarios. For example, let's take a look at the "`CatalogItem`" DTO:

```
namespace com.thecoderlife.example.productcatalog.dto
{
    /// <summary>
    /// Represents product data in the catalog
    /// </summary>
    [Serializable]
    public class CatalogItem
    {
        #region Properties

        public string ProductId { get; set; }
        public string Title { get; set; }
        public float ListPrice { get; set; }
        public string About { get; set; }
        public string MainPicture { get; set; }

        #endregion
    }
}
```

Picture 10: "CatalogItem" DTO class

It contains only the necessary data in order to build a product catalog list (in a web page, for instance). Once I modeled the "`Product`" class as an Aggregate root, it must be loaded as a whole from whatever the data source it is persisted. Because it was built to support and tackle business rules and complexities, its parts, individually, do not make sense.

Now imagine that you have thousands of products and some millions of users. Would you think it would be a good idea loading all "`Product`" instances for simply getting data in order to build and show a catalog of products? Definitely not.

The gist of this question is: when you don't need to treat any business rule (as when you just need to show some data), you should avoid the overhead that your Model persistence normally adds. That is why the method "`GetCatalogItem`" goes toward to the repository: it only needs data.

Running the Code Example

In this post, I have covered what I consider the most important concepts. However, if you want to explore a little bit more, the complete code example can be downloaded here. It is a VS 2012 Solution and contains a unit test project, where you can find some unit test methods that you can use for running and debugging and check how the business model has been tested.

Conclusion

In this complex universe of Software Engineering, Programing and Computer Science as well,& I tried to share with you guys part of my experience and knowledge regarding "What DDD is really about" and "How to approach your problem in a DDD fashion".

I really believe that no one is the owner of truth and the "trade of" law is always present. It is valid to keep yourself informed, reading books, articles, etc. Therefore, never do something just because everybody has been doing it. Never lose your critical sense. I hope it has been helpful for someone. :)

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 --