Useful Links

Sample code demonstrates DDD using entity framework.  
<http://blog.micic.ch/net/entity-framework-code-first-approach-and-domain-driven-design>  
<http://www.codeproject.com/Articles/56767/Domain-Driven-Design>  
<http://www.codeproject.com/Articles/56767/Domain-Driven-Design>

Ubiquitous language

Need for some common language shows up. Eric Evans calls this language as Ubiquitous Language. Your language is your model, this means that both Domain Experts and Developers should talk the same language which is build on the model. And changes to the Ubiquitous Language means changes to the Model. If Experts start using new terms this should be represented in your model, in your code, diagrams and speech.  
  
How could it look?  
\* Speech (This means daily conversations between team members.)  
\* Diagrams (Often these are quick diagrams on the whiteboard, which you created with your Experts.)  
\* Writing (This could be NOT long documents, describing the Model.)  
\* UML (But there is no place for cumbersome one.)

Eg. Customer can order..would mean a method orders on customer object..

### So Why “There Is No [Database](http://www.developerfusion.com/t/database/)”?

Domain Driven Design states specifically, in the name, why – we are designing our applications from the point of view of the **Domain**, and the **Domain** is drawn from the **Ubiquitous Language** we negotiate with our **Domain Experts**.

With persistence ignorance, we try and eliminate all knowledge from our business objects of how, where, why or even if they will be stored somewhere

Entities and Value ObjectsDDD has refined this concept a little, by splitting the idea of these business objects into two distinct types, **Entities** and **Value Objects**

<http://www.developerfusion.com/article/10077/domain-driven-design-a-step-by-step-guide-part-1/>

The key defining characteristic of an Entity is that it has an Identity – it is unique within the system, and no other **Entity**, no matter how similar is the same **Entity** unless it has the same **Identity**.

**Identity** can be represented in many ways on an **Entity** – it could be a numeric identifier (the classic CustomerID), it could be a Guid (the classic … oh never mind), or it could be a natural key (like the CustomerNumber your Customer was given by your CRM system when they first bought from you).  
So basically Entity is object that has **Attributes and Identity**.

Examples of common Entities are: Customer, Product, Container, Vehicle

*Example: Most airlines distinguish each seat uniquely on every flight. Each seat is an entity in this context. However, Southwest Airlines (or EasyJet/RyanAir for Europeans) does not distinguish between every seat; all seats are the same. In this context, a seat is actually a value object.*

### Value Objects

The key defining characteristic of a Value Objects is that it has no Identity. Ok, perhaps a little simplistic, but the intention of a **Value Object** is to represent something by it’s attributes only. Two **VOs** may have identical attributes, in which case they are identical. They don’t however have any value other than by virtue of their attributes

**Value Object are similar to the Entities expect that they do not need Identity,**.

Another aspect common to **VOs** is that they should probably be immutable, once created they cannot be changed or altered. You can create a new one, and as they have no identity, that is just the same as changing another one.

Examples of common Value Objects: Money, Address, ProductCode

Aggregates are groups of things that belong together. An Aggregate Root is the thing that holds them all together.

An **Aggregate** is slightly different, where as **Customers** and **Orders** can exist in the system independently, some **Entities** and **Value Objects** make absolutely no sense without their parent. The obvious example following on is **Orders** and **OrderLines**.

**OrderLines** have no reason to exist without their parent **Order**, nor can they belong to any other **Order**. In this case, **Order** and **OrderLines** would probably be an **Aggregate**, and the **Order** would be the **Aggregate Root**

The Root is an Entity, and it is the only object accessible from outside.

The main, and possibly obvious restriction on **Aggregate Roots** is, they must be **Entities**, and cannot be **Value Objects**. Back to the previous post, you will remember that **Entities** have **Identity**, and **Value Objects** do not – you could not ask a **Repository** to retrieve an **Aggregate Root \*if it had no \*Identity**.

### Factory: Creation of an object can be a major operation in itself, but complex assembly operations do not fit the responsibility of the created objects. This operations could be accomplished with [Factories](http://en.wikipedia.org/wiki/Factory_method_pattern). But not always it should be separate object. In scope of DDD Factory could even be a simple constructor. Two major requirements to the Factory are: Atomicity - this means that operation of creating an Entity should also create whole Aggregate associated with it. Abstracted to the type desired - this means that Factories should return you types rater than concrete classes. (i.e. Interfaces).

Repositoryis an object which provide you CRUD operations to your *Aggregates*.  
*Repositories* have many advantages, including the following:

* They present clients with a simple model for obtaining persistent objects and managing their life cycle.
* They decouple application and domain design from persistence technology, multiple database strategies, or even multiple data sources.
* They communicate design decisions about object access.
* They allow easy substitution of a dummy implementation, for use in testing (typically using an in-memory collection).

### ****Persistence Ignorance**** One of the key requirements to your Infrastructure to be able deal with DDD is Persistence Ignorance. This means that once you have your Infrastructure setuped you no longer need to care about how your data goes to database and how it is saved there.

While designing your architecture with DDD you often uses such patterns or approaches like:  
  
**SOA** ([Service-oriented architecture](http://en.wikipedia.org/wiki/Service-oriented_architecture))  
Generally speaking, SOA is a flexible set of design principles used during the phases of systems development and integration. It provides set of Services with which you are working.  
  
**AOP** ([Aspect-oriented programming](http://en.wikipedia.org/wiki/Aspect-oriented_programming))  
In computing, aspect-oriented programming (AOP) is a programming paradigm in which secondary or supporting functions are isolated from the main program's business logic. It aims to increase modularity by allowing the separation of cross-cutting concerns, forming a basis for aspect-oriented software development.  
  
**IoC** ([Inversion of Control and Dependency Injection](http://andriybuday.blogspot.com/2009/10/inversion-of-control.html))  
[Inversion of Control](http://en.wikipedia.org/wiki/Inversion_of_control) is principle with which you could easily resolve your dependencies in code delegating setup to some other part like IoC framework ([StructureMap](http://structuremap.sourceforge.net/Default.htm) for example).