Domain Driven Design, MVC & Entity Framework

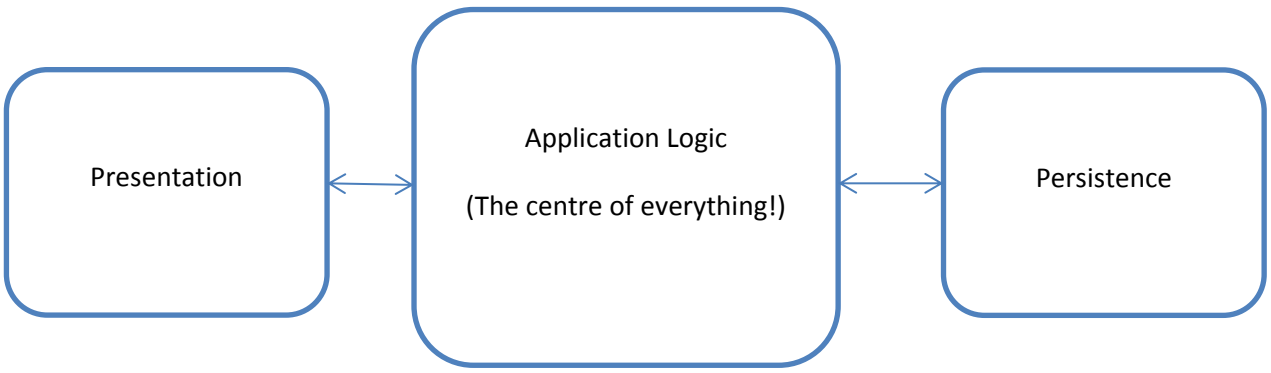
Domain Driven Design (DDD) is rather a new name for something old. Proper, object oriented code that focuses on the representation of the problem domain through Objects. MVC has also been around for a long time but it has become popular in Web Development only the last few years after Microsoft was forced to embrace it following the success of Ruby on Rails. Entity Framework is Microsoft's ORM (object-relational mapping) Framework and it has come years after NHibernate. I am not entirely sure why there is so much "wheel reinventing" out there. Perhaps we are in some sort of dark ages of computer programming methodologies and we need to keep going back to existing ideas and give them new names. DDD is nothing more than common sense and I don't think that any of its principles are radically new. I suspect that the answer to why there seems to be so much wheel reinvention is that simply the tools we had available to us until now were too awkward to work with and were created not always with the purest intentions in mind. ASP.NET Web Forms is a classic example. A whole "generation" of programmers has been educated to program with something that does not really fit the Web just because Microsoft has decided that Web programming should look like Windows programming. Thankfully, for the first time in a long time, it seems that we may eventually have a set of tools and technologies that allow us to focus on the problem at hand and its complexities instead of focusing on the complexities of the technologies we use to address the problem. In this document I am hoping to bring together DDD, MVC and Entity Framework and show you how easy it is today to build DDD code that puts the "Model" where it belongs, at the centre of our attention!

Every web project has 3 major, separate but communicating components to it and most programmers can intuitively understand them even if they don't always do their best to keep them as separate as they should.

These are:

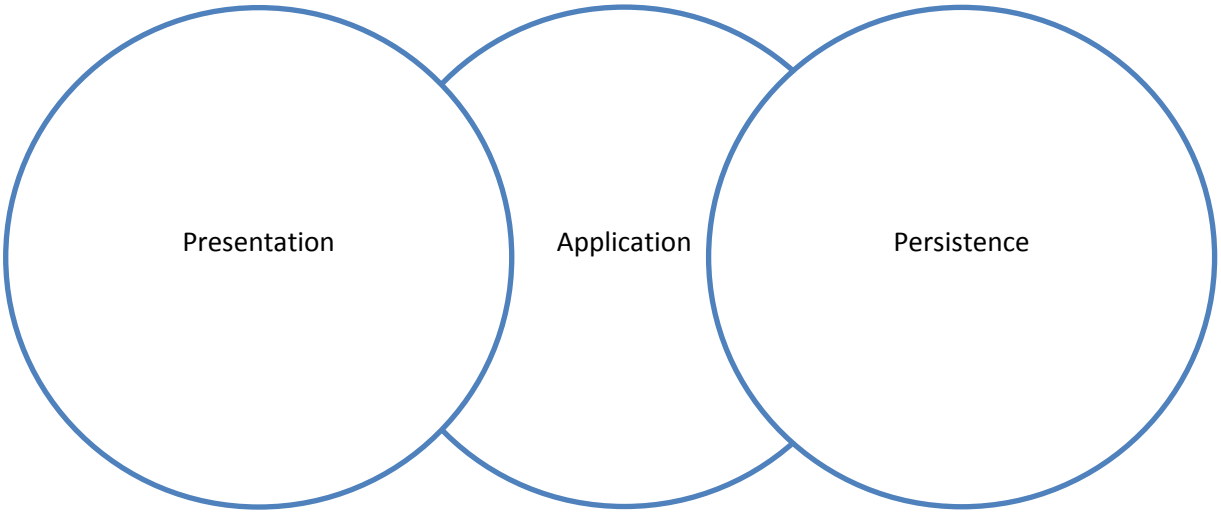
a) Presentation using Html, Css and Javascript. Interaction with web html web forms and http details
b) Application logic and code
c) Persistence logic and code (usually using a database)

In an ideal world you would quite naturally envision without having to think too deeply about it, just by having the most basic principles of computer programming in mind such as simplicity, clarity, separation of concerns and loose coupling that the flow of things should look something like this.

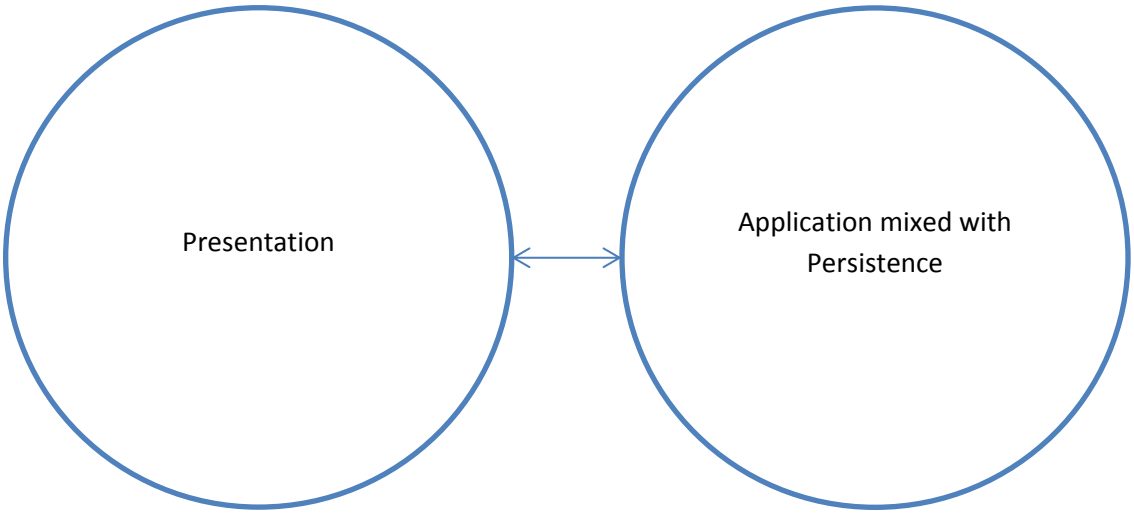


If you have done some Web development the last few years however, you will not be surprised to hear, that alarmingly enough, this diagram, simple and natural as it may look when you think about it, was not always how things actually worked on the Web. The reality of the situation is best described by the following 3 diagrams.

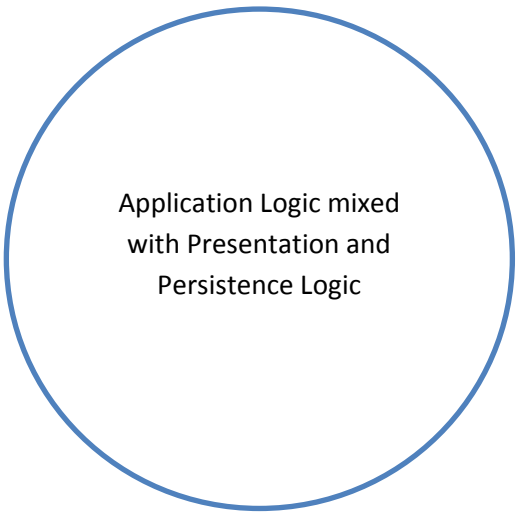
a) A typical ASP.NET Web forms application could easily follow this model



b) “Transaction Script” & “Active Record” Patterns follow this model



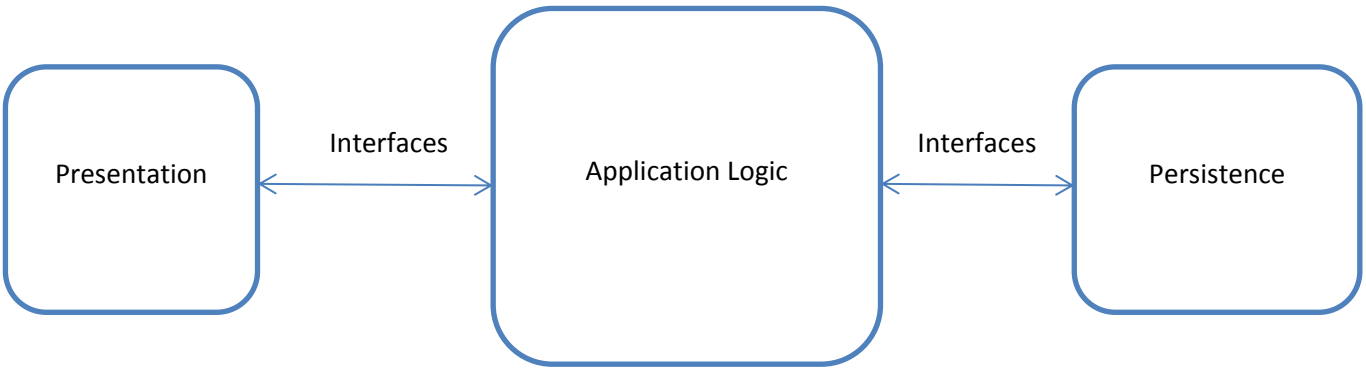
c) The worst case scenario!



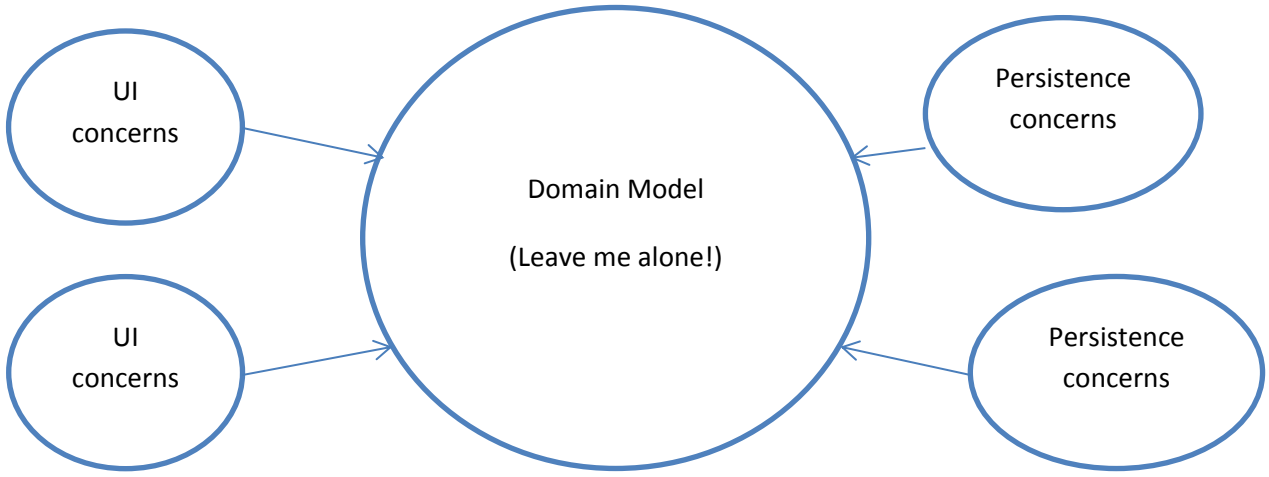
ASP.NET Web Forms made it very easy to create applications that followed models a and c. Your code behind or even your aspx page could very easily become the focus of control, the place where a mixture of presentation, application and persistence logic occurs to help you display your web pages. While that may not necessarily be a catastrophe for a lot of small and medium sized web applications, it can introduce a lot of problems with larger applications. The problems of this model are well known and are extensively documented. Just to give a brief summary I will mention the following

a) Unsuited for unit testing
b) Lack of transparency
c) Dynamically produced HTML/JavaScript not always suitable and not always following standards
d) Lack of control over HTML/JavaScript produced
e) Violating "Separation of Concerns" Principle
f) Awkward, "unnatural" & error prone page life cycle model
g) Viewstate "weight"

Of course it is not impossible to create proper web applications with ASP.NET Web Forms or PHP or any programming framework no matter how awkward the task may be. Most experienced programmers have been writing 3-Tier web applications that follow the "ideal" model depicted below



The only problem was that their task was made much more difficult than it should be because of the lack of the appropriate tools and frameworks. Interaction with the UI component was not always clear or easy resulting in the "pollution" of the "Application Logic Space" with unnecessary UI details. Interaction with the persistence medium was also not easy resulting in more "pollution", this time with Persistence details. The ideal world of DDD however is one where your Application Logic, or in DDD terminology, your Domain Model Code is clear of any UI or persistence details. Its sole purpose and focus is your Problem Domain and not any display or persistence concerns.



You may well argue at this point that yes, that all sounds good but…

a) a lot of web applications are just CRUD applications anyway where persistence and display is all there is and
b) is it really worth the trouble of going down the DDD route for small or even large projects?

Before I give the answers that I think are correct, let me state the obvious, that when it comes to programming methodologies and patterns there is no absolute truth, no answer suitable for all intents & purposes. With this in mind my answer is the following

a) If your focus is on CRUD applications then you may need to start looking for a new job because with the pace that ORMs and "Designer" applications move, I suspect that there will not be much left for you to do in a few years time!
b) It is true that DDD is at its best when it addresses complex problem domains that may require hundreds of different objects. However don't think that you cannot apply it to smaller projects and learn about it. If you use the right tools such as MVC 3 and Entity Framework 5, then a "DDD compliant" implementation is easily within your grasp even if it is not perfect. Going down the DDD route will help you write better, cleaner code that actually makes sense and in the long or even short term makes your life a lot easier.

I have to admit myself that the pace of progress in our industry can be a bit scary at times. Things that took you a long time to complete and were considered part of your skillset such as writing sql and stored procedures are being replaced by Ling to Sql and Entity Framework. JQuery and MVC make front end development a lot easier than it used to be. However this is not something to be afraid of if you want to remain successful in this business. Instead of seeing it as something that "reduces" your value as a developer, you should see it as something that liberates you and provides you the tools to write truly great software for increasingly complex problem domains and that is where DDD comes in. DDD and other techniques and patterns will help us build the software of the future, software whose codebase reflects the problem domain and is concerned with the problem domain and not with superficial technical details and that is exactly what software should be all about.

Before I explain DDD the way that I see it I think it will be of benefit if I provide a brief overview of the 2 patterns mentioned before, namely the Transaction Script pattern and the Active Record pattern and also explain how MVC and Entity Framework help DDD.

Transaction Script

I think most programmers know about this pattern and have used it even if they were not aware of its name. It's the simplest pattern and probably the first that comes to the mind of an inexperienced developer or a developer who simply wants to build a small application quickly. Let us assume that we have a simple, classic online shopping application that allows users to add items to their "virtual shopping baskets". If we are about to follow the Transaction Script pattern we will create a "BasketManager" or "BasketProcessor" or whatever you may want to call it class that is responsible for everything related to the virtual shopping basket.

```
public class BasketManager
{
        public AddToBasket(Product product, int quantity)
        { … }
        public RemoveFromBasket(Product product)
        { … }
        public SaveBasket(Basket basket)
        { … }
        public DeleteBasket(Basket basket)
        { … }
```

```
        …
}
```

Notice how the BasketManager class does all the work of adding/removing items, persisting the basket where it may need to be persisted and so on. While structure is always better than no structure and the Transaction Script does provide a basic structure that is easy to understand, it does not upscale very well. Not only it gives too much responsibility to a single class resulting in a long, error prone, full of methods class which is not really object oriented design but it also makes the interaction between different areas of the application quite difficult since manager classes will have to interact with other manager classes in a rather chaotic way. Interestingly enough, you can probably improve the situation by using the DDD (and common sense) idea of "aggregate roots" (more on this later).

Active Record

Active Record is a "data centric" pattern that sees your objects from the "persistence point of view". While this may be one of its inherent weaknesses, Active Record is a very useful pattern very well suited to the majority of web applications out there. Typically an object will correspond to a table in your database and it will be responsible for its own persistence which is the big difference with DDD. Going back to the online shopping application we would have

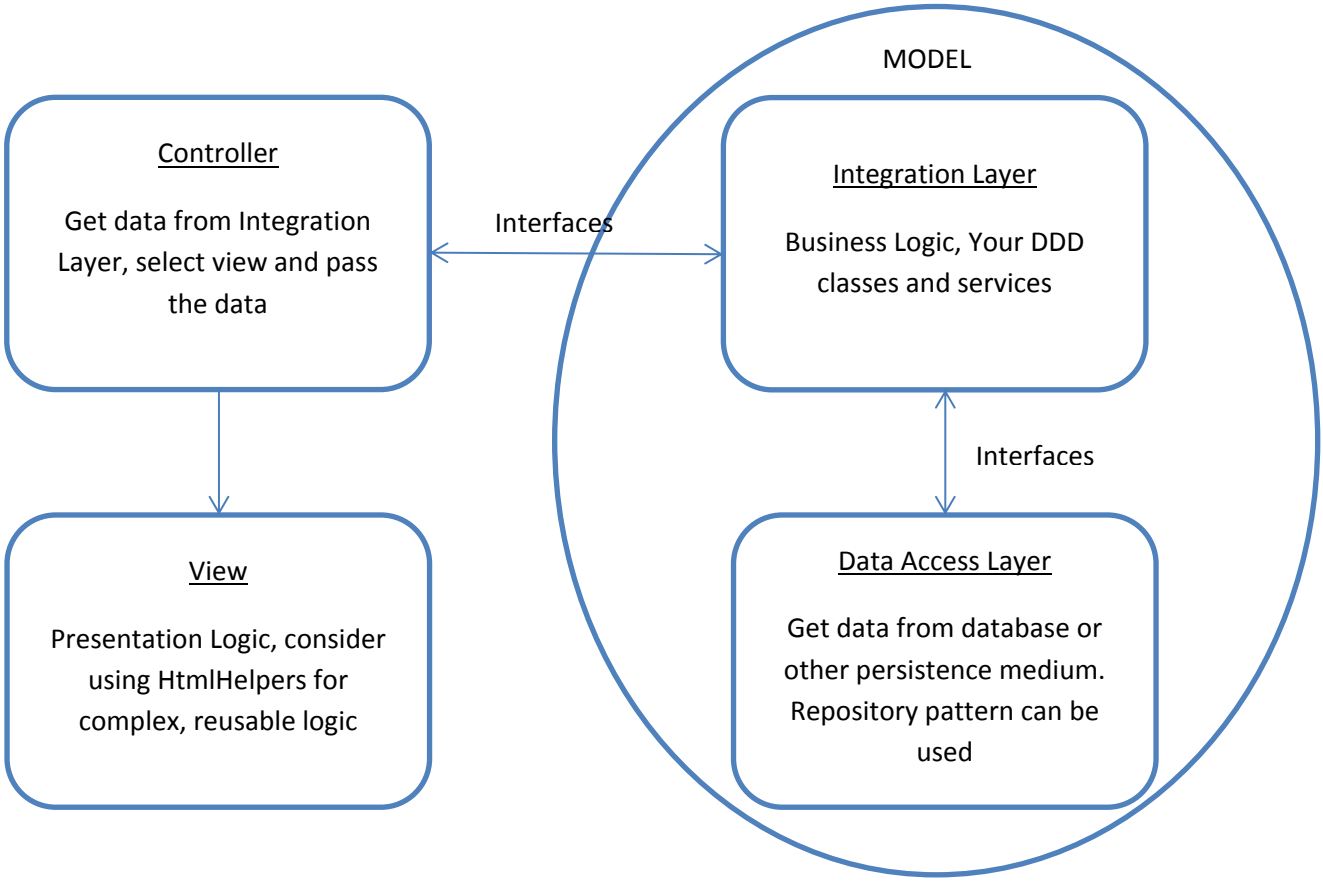
```
public class Basket
{
        public int BasketId {  get;  set;  }
        public List<Product> Products {  get;  private set; }
        public void AddProduct(Product product, int quantity)
        { … }
        public void Save()
        {
                dbAccess.SaveToDatabase(this);
        }
        public static Basket Load(int basketId)
        {
                dbAccess.GetFromDatabase(basketId);
        }
        …
}
```

Notice how the Basket is now responsible for a lot of the application logic that was before part of the BasketManager. Also notice how the object is responsible for persisting/loading itself to/from the database. This constitutes a big improvement. Here is a proper object that encapsulates its data and operations in a way that makes sense and is proper object oriented code. In my experience Active Record can be applied to the vast majority of web applications out there and it will deliver a proper structure that allows for the creation of proper object oriented code. Its weakness is that it mixes application logic with persistence which may not always be such a good idea and it can also be awkward to work with in situations where your domain model, the objects that express the concepts and problem you are trying to resolve do not have corresponding, easily identifiable entities in your database or other persistence medium. Once again, the application of the "aggregates root" idea of DDD can help you structure Active Record code in a better way.

MVC and how it helps DDD

MVC is great for building N-Tier applications since it has built in support for the Model-View-Controller pattern. If you use MVC properly (as a rule of thumb, check out your Controllers code, if action methods have more than 10-20

lines of code then you are not using MVC properly, you just use MVC the DLL to abuse MVC the pattern). As a summary of how MVC applications should ideally be structured, please have a look at the following diagram.



As you can see the Model in MVC includes the "Application Logic" & "Persistence Logic" that we usually find in 3-Tier web applications. This is where your DDD code will happily live when you build an MVC/DDD application. That is not however the only convenience that MVC offers. One of the handiest features of MVC is its model binding and model validation techniques and also its support for strongly typed views.

Model Binding

When you pass data to an MVC application requesting one of its pages, MVC does its best in the background to match the raw data that you pass, to valid objects that you the programmer can define. As an example if you have a class Person

```
public class Person
{
        public int Age { get; set ; }
        public String Name { get; set; }
}
```

and then you have in your controller a method to Update this person

```
public ActionResult Update(Person p)
{ … }
```

Now, when in your html form you pass age and name using the appropriate naming convention, MVC will match that data with the Properties Age and Name and create a valid Person object for you.

Model Validation

There are a few different approaches here but the ones we are interested in are two.
a) Using attributes

b) Using the IValidatableObject interface

What that means is that your Model Classes can be responsible for their own validation without you having to write lots of lines of ugly, awkward looking validation code. What is even better is that if you use Entity Framework, it can also use that validation information for its own puproses and validate your objects before inserting them to the database. As a simple example of how validation code looks like

```
//Using Attributes
public class Person
{       [Required]
        public Int Age { get; set ; }

        [Required]
        public String Name { get; set; }
}

//Using IValidatableObject
public class Person
{
        public Int Age { get; set ; }
        public String Name { get; set; }



        public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
        {
                List<ValidationResult> errors = new List<ValidationResult>();

                if (Age == null)
                   errors.Add(new ValidationResult("*", new string[] { " Age " }));
                if (String.IsNullOrEmpty(Name))
                   errors.Add(new ValidationResult("*", new string[] { " Name " }));

                return errors;
         }
}
```

Strongly Typed Views

With strongly typed Views, your View is bound to a Model class (or even collections of Model classes) and functionality for producing the correct Html to work with this Model is provided. Access to the Model's public properties and methods is also very easy. As a result you can write something as simple as this

```
@Html.TextBoxFor(p => p.Name)

@Html.TextBoxFor(p => p.Age)
```

You can now rest assured that if you submit this form, MVC will be able to bind the data you pass to the Name and Age properties of a Person object.

These 3 attributes of MVC are extremely important. Not only they allow you to write much shorter and much cleaner code than ever before, but they clearly demonstrate that the Model is at the heart of MVC. Your HTML/HTTP interactions can result in the construction of valid model classes, these classes are then processed by your "Domain Model Code" and they can also be persisted to your data store. The 3 layers of your application can now easily work by exchanging objects.

Entity Framework and how it helps DDD.

Entity Framework is Microsoft's ORM. After a few years of perhaps not so successful releases, Entity Framework 5 seems to be very close to providing a fully DDD compliant Data Access Model. With Entity Framework 5 you can focus on writing "POCOs" (Plain Old CLR Objects), classes that are free of any of the cluttering data access details of EF5's predecessors such as earlier versions of EF or Ling to Sql. It is true that not all is perfect with EF5 but the progress made is definitely phenomenal and one can only expect that whatever shortcomings remain will be removed in subsequent releases.

Entity Framework lives in the EntityFramework.dll which includes the major new features such as the DbContext API and the "Code First" functionality. Some of it however also lives in .NET, mainly the System.Data.Entity dll.

"Code First" allows you to write your code first without even needing a database. If you want, it can create it for you or you can map your POCOs to an existing database. Just like MVC, Entity Framework uses "convention over configuration". Based on certain conventions that you follow, it will come to certain conclusions about the structure of your database. A property of PersonId in class Person will be assumed to correspond to a primary key called PersonId in your Person table in the database. If conventions are not good enough for what you are doing you can use configuration by applying data annotation attributes to the properties and fields of your classes or by using the "Fluent API" that allows you to make configurations in code.

Just like MVC, EF5 with Code First is focused on the Model, making your life a lot easier when it comes to persistence.

Domain Driven Design Principles

Let's start with some basic definitions that are perhaps oversimplified but will help to get us thinking in the right direction.

"Domain": The problem at hand and all the information related to it

"Domain Model": A group of concepts/entities (you could think of them as objects most of the time) used to describe the Domain

DDD is a collection of practises and patterns for structuring your object oriented system. The most important concepts of it are explained below.

Use proper terminology

When designing an application and creating its domain model you should use clear language and terms that both business and technical people can understand. You should always try to use existing business terminology when you can and avoid reinventing the wheel. When business terminology changes you should do your best to update your application so that it reflects these changes. DO NOT underestimate the importance of this! Good names drive good projects and bad names destroy them. There is nothing worse than a name that is conceptually inappropriate or

even worse contradictory to the reality of your business. Not only people may assume the wrong thing about what your code means or does but they are very likely to generate even more bad naming conventions.

Aggregates & Simplification

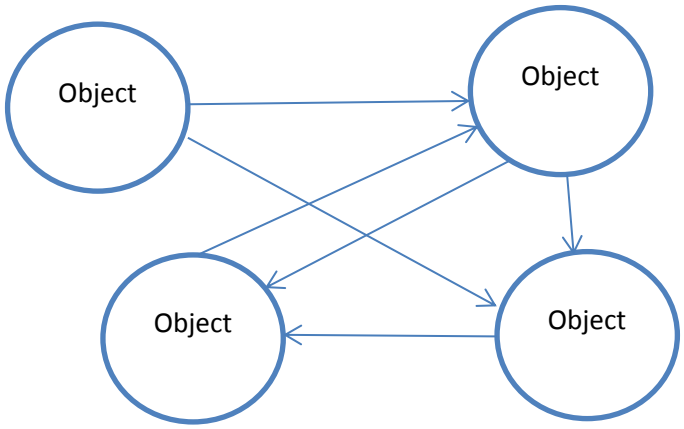
Let's take as an example a simple customers & orders application. A customer can multiple customer addresses. A customer can place orders. Orders consist of different order items corresponding to products. Let s forget about products and other details for now and focus on orders and customers.

We have 4 domain objects in our system

- Customer
- CustomerAddress
- Order
- OrderItem

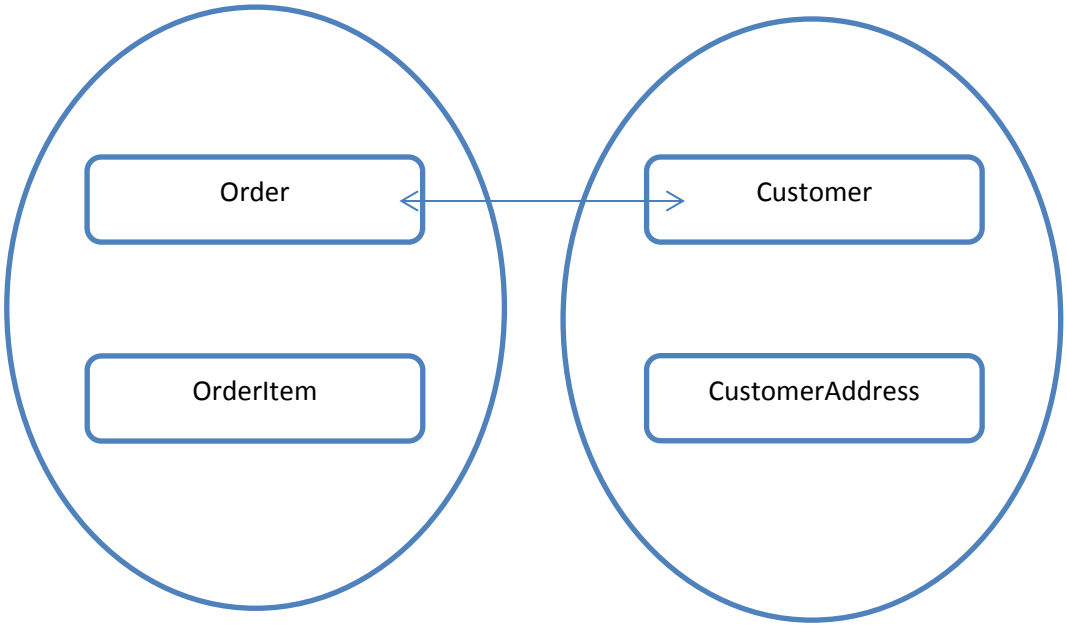
A formal definition of what an aggregate is, is the following: *"A cluster of associated objects that are treated as a unit for the purpose of data changes"*. The aggregate root is simply the object that is in "control" of this cluster and is responsible for coordinating actions and interactions of the objects in the cluster. Objects outside the cluster can only hold a reference to the aggregate root of the cluster.

That may all sound a bit too theoretical so let's see what problems it tries to address. A complex object oriented system will consist of a large number of objects. The complexity of the system usually arises not from the inner complexities of the code in each individual class but from the required interactions between the classes. This is actually a problem not only in computer systems but in any system.

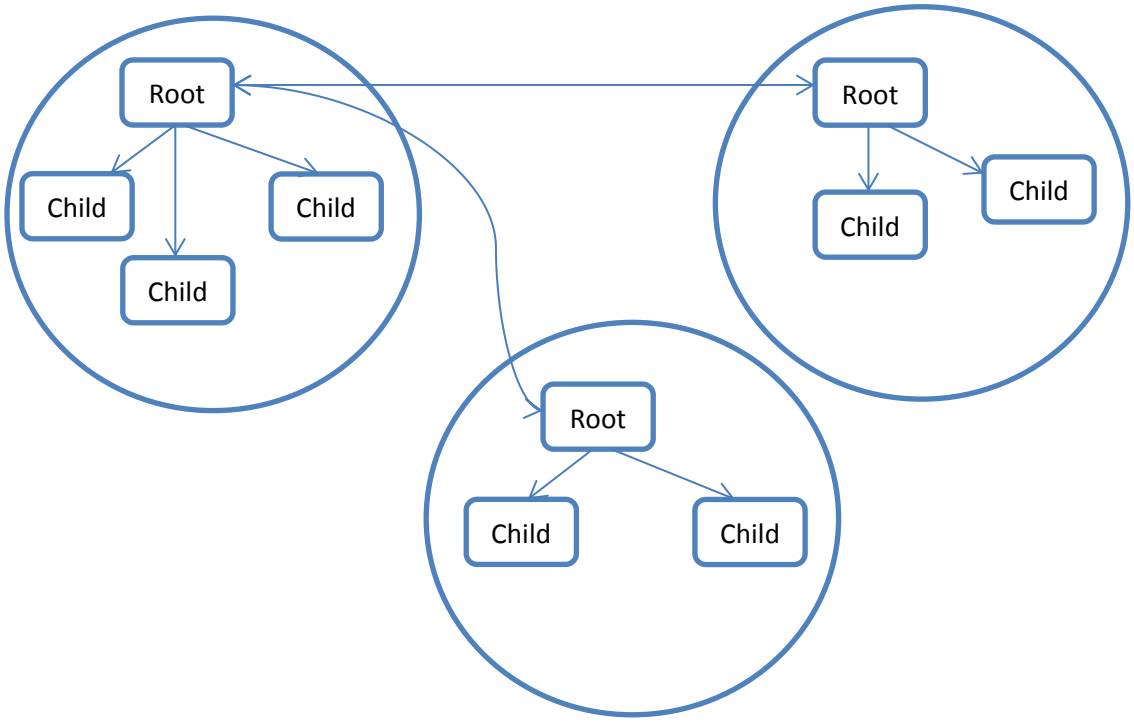


As you can see the number of possible interactions can easily become quite chaotic even if you only have 4 classes in your system. To avoid this problem, you need to define your clusters and ensure that the interactions between the objects follow a simple and meaningful paradigm.

For our simple example, it is quite obvious that there are 2 clusters, one for Customers and one for Orders. The aggregate root for the Customers cluster is the Customer class and the aggregate root for Orders is the Order class.



It is clear from the diagram above that this very simple structure results in a much cleaner model where the Customer class can interact directly and hold references to the Order class and vice versa. There needs to be no interaction between Customer and OrderItems or between CustomerAddress and Order and so on. In more general terms, if you think of the aggregate root as the root of a tree, the roots interact only with other roots or with their own children.



Below is a simple example of how your Order class may look like. Notice how all the business logic related to the Orders cluster needs to be present in this class (in our simple example that is just adding/removing order items), how a reference is held to the root of the Customers cluster and that there is no persistence code.

```
public class Order
{
	public Customer { get; set; }
	public int OrderId { get; set; }
	public decimal OrderValue { get; set; }
	public ReadOnlyCollection<OrderItem> OrderItems { get; private set; }
	public void AddItem(OrderItem item)
	{ … }
	public void RemoveItem(OrderItem item)
	{ … }
}
```

Services

You can think of services as classes that don't represent an entity in your application such as a customer or order but represent a process or activity which may or may not correspond to a business process. Usually services need to use the functionality of many different clusters and root classes. As an example think of an order service that has a method looking something like this

```
public void ProcessOrder()
{
	//get order details from basket, check if all items are still in stock & create order
	…

	//process payment with payment gateway
	…

	//if payment successful, save order
	…

	//email user
	…

	//log order in some sort of system log
	…
}
```

As you can see there are many different things that this method needs to do which are not necessarily strongly tied to the Order object. The service needs to delegate to Order some of its work, the Service should not do work that the Order class can do, it is there more as a coordinator of different classes and interactions, it is not there to add or remove order items from the Order. Also notice that given the fact the DDD objects are persistent ignorant, persistence is usually handled by Service classes, not directly but by using Repositories. It is true that if you use the Active Record Pattern or simply think that you can have a ProcessOrder method inside the Order class itself and let Order handle its own persistence you can still write some pretty decent code. I do think however that adding these details to your Order class may mean that you try to do too much in it and have to pass to it all sorts of other objects such as an object for sending/saving an email, an object that does the interaction with the payment gateway, one that logs the order to a log. All that detail means that your Order object is not focused any more on the plain, simple, business actions it needs to accomplish and begins to concern itself with details such as the email service used, the payment gateway etc. That lack of simplicity and clarity at the heart of your domain model may be detrimental to the overall system architecture.

Repositories

Persistence code should not be part of your domain model. You should define repositories that are responsible for all the data access needs of an object cluster. All data access logic is encapsulated in Repositories and your application should NOT use any other way of accessing the database. The Domain Model should call the methods of the Repository. The convention is to define a separate Repository for each aggregate root. In our example we should have 2 repositories, one for Customers and one for Orders.

Let's see 2 simple examples, one that uses a Repository and one that does not.

```
//Nasty code without a repository
public class HomeController : Controller
{
public ActionResult Index()
        {
                var dataContext = new OrderDataContext();
                var orders = from o in dataContext.Orders
                select o;
                return View(orders);
        }
}

//A better way
public class OrderRepository : IOrderRepository
{
        private OrderDataContext _dataContext;
        public OrderRepository()
        {
                _dataContext = new OrderDataContext();
        }

        #region IOrderRepository Members
        public IList<Order> ListAll()
        {
                var orders = from o in _dataContext.Orders
                select o;
                return orders.ToList();
        }
        #endregion
}

//Controller using Repository
public class OrdersController : Controller
{
        private IOrderRepository _repository;
        public OrdersController() : this(new OrderRepository())
        {}

        public OrdersController(IOrderRepository repository)
        {
                _repository = repository;
        }

        public ActionResult Index()
```

```
        {
                return View(_repository.ListAll());
        }
}
```

Build Loosely Coupled Components

Different layers of the application should expose functionality through interfaces. Why is this important? Let's look at a simple OrderProcessor class.

```
public class OrderProcessor
{
        private order_db = new order_accessor();
        public ProcessOrder(order o)
        {
                …..
                order_db.SaveOrder(o);
        }
}
```

OrderProcessor is now strongly tied to the order_accessor that is in our data access layer. If someone wants to use this class to save orders to a text file or submit it as xml to a webservice he won't be able to do it even if the operations he wants to perform are the same and defined below

```
interface IOrderRepository
{
        void SaveOrder(Order o);
        Order GerOrder(int orderId);
}
```

Let 's rewrite the OrderProcessor class with some simple changes that make it a lot better!

```
public class OrderProcessor
{
        private readonly IOrderRepository _orderRepository;
        public OrderProcessor(IOrderRepository orderRepository)
        {
                _ orderRepository = orderRepository;
        }
        public OrderProcessor() : this(new OrderRepository())
        {}

        …
}
```

This class is now much more flexible as it allows the OrderProcessor to be instantiated with a different implementation of the data accessor which could be a real implementation or a mock object used for unit testing. In general, when it comes to MVC applications with or without DDD, Controllers should know as less as possible about the Model and the Model should know as less as possible about the Data Layer.

- Define the contract (Interface) of the Controller to Model Interaction
- Define the contract (Interface) of the Model to Data Layer Interaction

Use Dependency Injection and Inversion of Control

For some bizarre reason, a lot of text books, tutorials and blogs out there present Dependency Injection like it is some overly complicated, esoteric topic and it takes them pages upon pages to explain it. Well, you ve got to sell your book somehow I suppose. Anyway, while there may be a bit of detail to Dependency Injection it can easily be explained in a few sentences. Look at the class below.

```
public class OrderProcessor
{
        private readonly IOrderRepository _orderRepository;
        public OrderProcessor(IOrderRepository orderRepository)
        {
                _ orderRepository = orderRepository;
        }
        public OrderProcessor() : this(new OrderRepository())
        {}

        …
}
```

This class already uses DI. As we have already discussed, this class does not have a hard coded dependency on a specific implementation of OrderProcessor. Instead, it expects to receive an appropriate object that implements IOrderRepository at runtime. The ***"dependency"*** of the Order Processor to an OrderRepository is "***injected"*** at runtime. If we wish to allow the OrderProcessor to be instantiated with a different implementation of the IOrderRepository, we can easily do so by passing a different concrete implementation class to its constructor. Using the constructor to achieve DI is not the only possibility, a setter property/method could be used or in more advanced scenarios the dependency can be injected by another factory class or tool.

Now that we are done with dependency injection let's have a look at Inversion of control. Let's assume that we have a class called OrderProcessorClient that needs to use our OrderProcessor. This class can instantiate the OrderProcessor by using

```
var myProcessor = new OrderProcessor();
```

OR

```
var orderRepository = new OrderRepository();
var myProcessor = new OrderProcessor(orderRepository);
```

There is nothing wrong with using these methods, in fact I think that for the vast majority of web projects this level of architecture is more than enough if all was done as it should (which is rarely the case of course). However, if you want to take things one step further you can still do a couple of things to improve this code. The problem with the code above is that it requires the OrderProcessorClient class to be aware of the dependencies of the OrderProcessor. If the client wants to pass a specific implementation of the OrderRepository other than the default that the parameterless constructor provides, it needs to create the repository and pass it to the constructor. That now creates a compile time dependency between the client class and the concrete repository implementation. In an ideal world the client shouldn't have to worry about such details.

The "classic" approach of resolving issues like these was using the factory pattern which provides for a factory class responsible for creating objects. In its simplest form it would be something like

```
public static class MyFactory
{
```

```
        public static OrderProcessor CreateOrderProcessor()
        {
                var orderRepository = new OrderRepository();
                var processor = new OrderProcessor(orderRepository);
                return processor;
        }
}
```

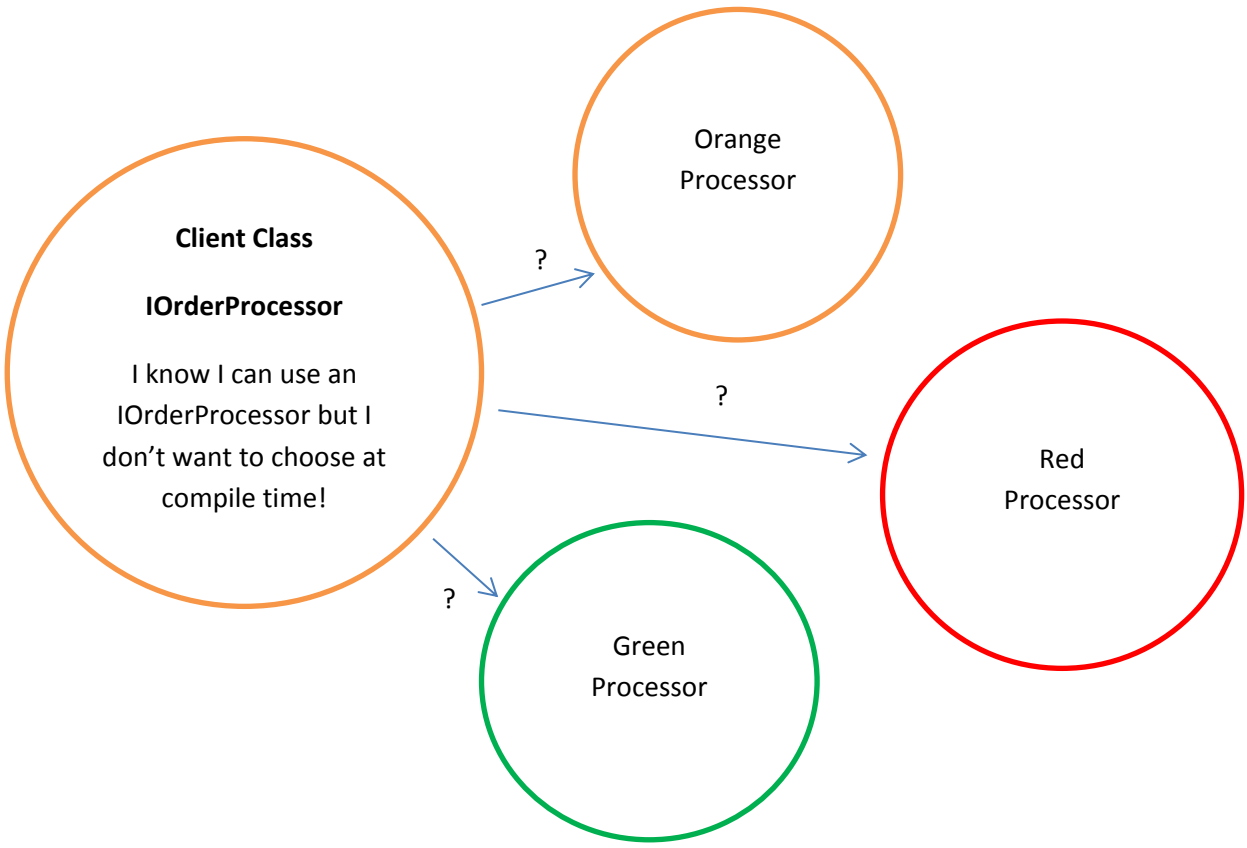
And now the OrderProcessorClient can simply call

```
var myProcessor = MyFactory. CreateOrderProcessor();
```

This solution is perhaps somewhat better but as you can probably guess the real problem is still there and the real problem is that you still have a dependency in your code to a concrete implementation of OrderRepository at compile time. If you need to create an OrderProcessor that uses a different implementation of the IOrderRepository interface you need to create a new method such as

```
public static OrderProcessor CreateDifferentOrderProcessor()
{
        var orderRepository = new SomeOtherRepository();
        var processor = new OrderProcessor(orderRepository);
        return processor;
}
```

There is probably no good enough solution for this problem using “traditional” programming techniques, no matter how you structure your code you will never have a perfect solution simply because what is required is a mapping between an interface (or abstract class) and a concrete implementation. What you would ideally like to have is the ability to write code that uses Interfaces and chooses concrete implementations at runtime.



Thanks to more powerful and faster computers and sophisticated software platforms such as .NET it is now possible to use Reflection and Metadata to do a lot of the injection of dependencies at runtime. An IoC container (Inversion of Control Container) is the software tool that allows you to do such a mapping in code or by using an xml configuration file. The name "Inversion of Control" comes from the fact that your client class is not responsible now for creating an OrderProcessor or for deciding which concrete implementation it needs. The IoC container is the one in control of this process. As a simple example of how this would look like in code if you used a popular IoC container such as Ninject.

```
// Instruct IoC to use SomeOtherRepository in place of IOrderRepository
Bind<IOrderRepository>().To<SomeOtherRepository>();
…

// Get an instance of the IoC
IKernel kernel = new StandardKernel();

// Get an instance of an orderProcessor, IoC will check its mappings and create it
// by using SomeOtherRepository
var orderProcessor = kernel.Get<OrderProcessor>();
```

DDD, TDD & Unit Testing

Another 3 letter word that is very popular these days is TDD or Test Driven Development. You will be happy to know that DDD and TDD are perfect for each other. Despite the fact that they can probably be considered two different ways of building applications since TDD is really about designing software more than it is about testing, they can produce very similar results and perhaps they can be combined when required. When you use TDD you are producing testable functionality in small steps focusing on the problem domain. UI and persistence concerns do not enter your world just like in DDD. If you need to include UI, persistence or perhaps some other external concern you do so by providing hard coded or mock objects that simulate real life behaviour. TDD promotes separation of concerns and to be able to use it effectively, you need to be assured that you have the right tools and the right system architecture that will allow you to focus on the problem domain. The even better news is that whichever of the 2 you use you can be assured that it will be easy to create unit tests. With TDD, these will be written during application development. With DDD they may be written afterwards or perhaps you may choose to use a combination of DDD & TDD where DDD is used for the overall system design and architecture and TDD is used to slowly fill in the details. One way or the other, I think that it is important that two of the most predominant software development methodologies today have a lot in common and we can use this to our advantage. There is one thing we know for sure: No problem domain and business situation is the same and no software construction methodology, no matter how comprehensive and solid it is, can suit all the different environments out there. So it is of some importance to know that DDD & TDD are not mutually exclusive, they can produce similar results, they can be combined and they both make the creation of unit tests easy.

A practical example

As a sample implementation of all the ideas presented in this document (with the exception of using an IoC container) I will include here some code that I have been working on for a small online Eshop application. The code includes 2 POCO classes for an Order & OrderProduct, the OrderRepository class, a simple OrdersController and the DbContext class used for accessing the database. The code uses “Dynamic Query”. More info about dynamic query on

http://weblogs.asp.net/scottgu/archive/2008/01/07/dynamic-linq-part-1-using-the-linq-dynamicquery-library.aspx

EShopData Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using EShop.Models.DomainModel;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace EShop.Models.DataAccess
{
    public class EShopData : DbContext
    {
        public DbSet<ApplicationLog> ApplicationLogs { get; set; }
        public DbSet<Basket> Baskets { get; set; }
        public DbSet<BasketProduct> BasketProducts { get; set; }
        public DbSet<Brand> Brands { get; set; }
        public DbSet<CreditCard> CreditCards { get; set; }
        public DbSet<Customer> Customers { get; set; }
        public DbSet<CustomerAddress> CustomerAddresses { get; set; }
        public DbSet<EmailSent> EmailSents { get; set; }
        public DbSet<Invoice> Invoices { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<OrderProduct> OrderProducts { get; set; }
        public DbSet<Product> Products { get; set; }
        public DbSet<ProductCategory> ProductCategories { get; set; }
        public DbSet<ProductImage> ProductImages { get; set; }
        //public DbSet<ProductToCategory> ProductToCategories { get; set; }
        public DbSet<Review> Reviews { get; set; }
        public DbSet<Supplier> Suppliers { get; set; }
        public DbSet<StaticPage> StaticPages { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            // Configure Code First to ignore PluralizingTableName convention
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

            // Turn off lazy loading to make sure ignorance does not lead to hidden DB calls!
            //this.Configuration.LazyLoadingEnabled = false;

            // Invoice primary key is invoice number
            modelBuilder.Entity<Invoice>().HasKey(t => new { t.InvoiceNumber });

            // OrderProduct composite key
            modelBuilder.Entity<OrderProduct>().HasKey(t => new { t.OrderId, t.ProductId });

            // BasketProduct composite key
            modelBuilder.Entity<BasketProduct>().HasKey(t => new { t.BasketId, t.ProductId });

            modelBuilder.Entity<Product>()
            .HasMany(t => t.Categories)
            .WithMany(t => t.Products)
```

```
            .Map(m =>
            {
                m.ToTable("ProductToCategory");
                m.MapLeftKey("ProductId");
                m.MapRightKey("CategoryId");
            });
        }
    }
}
```

OrderProduct Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace EShop.Models.DomainModel
{
    public partial class OrderProduct
    {
        public OrderProduct() { }

        public int OrderId { get; set; }
        public int ProductId { get; set; }
        public int Quantity { get; set; }
        public decimal Price { get; set; }

        public Order Order { get; set; }
        public Product Product { get; set; }
    }
}
```

Order Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Collections.ObjectModel;
using EShop.Models.Helpers;

namespace EShop.Models.DomainModel
{
    public partial class Order
    {
        public Order()
        {
            this.Products = new List<OrderProduct>();
        }

        public int OrderId { get; set; }
        public int CustomerId { get; set; }
        public int OrderStatus { get; set; }
        public OrderStatusEnum OrderStatusAsEnum
        {
            get
            {
                return (OrderStatusEnum)Enum.Parse(typeof(OrderStatusEnum), OrderStatus.ToString());
            }
            set
            {
                OrderStatus = (int)value;
```

```
    }
}
public decimal DeliveryCharge { get; set; }
public DateTime Created { get; set; }
public DateTime Modified { get; set; }
public DateTime? SentToWarehouse { get; set; }

public Customer Customer { get; set; }
public ICollection<OrderProduct> Products { get; private set; }
public int? InvoiceNumber { get; set; }
public Invoice Invoice { get; set; }

public void AddProduct(Product product, int quantity)
{
    if (product == null)
        throw new ArgumentException("Product is null");
    if (quantity <= 0)
        throw new ArgumentException("Invalid quantity");

    var orderProduct = new OrderProduct()
    {
        OrderId = this.OrderId,
        Price = product.Price,
        Quantity = quantity,
        ProductId = product.ProductId
    };
    Products.Add(orderProduct);
}

public void RemoveProduct(Product product)
{
    if (product == null)
        throw new ArgumentException("Product is null");

    var productToRemove = Products.SingleOrDefault(p => p.ProductId == product.ProductId);
    if (productToRemove != null)
        Products.Remove(productToRemove);
}

public void GenerateInvoice()
{
    if (this.Invoice == null)
    {
        this.Invoice = new Invoice()
        {
            TotalValue = this.TotalAmount,
            Created = DateTime.Today,
            Modified = DateTime.Today
        };
    }
}

public void CancelOrder()
{
    this.OrderStatusAsEnum = OrderStatusEnum.Cancelled;
}

public void MarkOrderAsSentToWarehouse()
{
    this.OrderStatusAsEnum = OrderStatusEnum.SentToWarehouse;
    this.SentToWarehouse = DateTime.Now;
}

public void MarkOrderAsFullFilled()
{
    this.OrderStatusAsEnum = OrderStatusEnum.FullFilled;
```

```
        }

        public static Order CreateOrderFromBasket(Basket basket)
        {
            if (basket == null)
                throw new ArgumentException("Basket is null");
            if (basket.BasketProducts == null || basket.BasketProducts.Count == 0)
                throw new ArgumentException("Basket is empty");

            Order order = new Order()
            {
                CustomerId = basket.CustomerId.HasValue ? basket.CustomerId.Value : 0,
                OrderStatusAsEnum = OrderStatusEnum.New,
                SentToWarehouse = null,
                Created = DateTime.Now,
                Modified = DateTime.Now
            };

            foreach (var basketProduct in basket.BasketProducts)
                order.AddProduct(basketProduct.Product, basketProduct.Quantity);

            return order;
        }

        public decimal MinimumOrderValue
        {
            get { return ConfigHelper.MinimumOrderValue; }
        }

        public decimal ProductsPrice
        {
            get { return Products.Sum(p => p.Product.Price * p.Quantity); }
        }

        public decimal TotalAmount
        {
            get
            {
                return ProductsPrice + DeliveryCharge;
            }
        }
    }
}
```

BaseRepository Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Objects;
using System.Linq.Expressions;
using System.Web.Helpers;
using System.Linq.Dynamic;
using EShop.Models.DataAccess;
using EShop.Models.Helpers;

namespace EShop.Models.Repositories
{
    public class BaseRepository  : IDisposable
    {
        protected EShopData dbContext;

        protected BaseRepository(EShopData db)
        {
```

```
        this.dbContext = db;
    }

    protected IQueryable<TEntity> GetPagedAndSorted<TEntity>(IQueryable<TEntity> query,
SortAndPageCriteria sortAndPage, out int count)
    {
        //SORTING
       string direction = sortAndPage.SortDirection == SortDirection.Descending ? "DESC" :
         "ASC";

        query = query.OrderBy(String.Format("{0} {1}", sortAndPage.SortBy, direction));

        //GET COUNT
        count = query.Count();

        //PAGING
        int startRowIndex = (sortAndPage.PageNumber - 1) * sortAndPage.PageSize;
        query = query.Skip(startRowIndex).Take(sortAndPage.PageSize);

        return query;
    }

    protected IQueryable<TEntity> GetPagedAndSorted<TEntity>(IQueryable<TEntity> query,
SortAndPageCriteria sortAndPage)
    {
        //SORTING
        string direction = sortAndPage.SortDirection == SortDirection.Descending ? "DESC" :
          "ASC";

         query = query.OrderBy(String.Format("{0} {1}", sortAndPage.SortBy, direction));

        //PAGING
        int startRowIndex = (sortAndPage.PageNumber - 1) * sortAndPage.PageSize;
        query = query.Skip(startRowIndex).Take(sortAndPage.PageSize);

        return query;
    }

    protected IEnumerable<TEntity> Get<TEntity>(IQueryable<TEntity> query,
Expression<Func<TEntity, bool>> filter = null, Func<IQueryable<TEntity>,
IOrderedQueryable<TEntity>> orderBy = null)
    {
        if (filter != null)
        {
            query = query.Where(filter);
        }

        if (orderBy != null)
        {
            return orderBy(query).ToList();
        }
        else
        {
            return query.ToList();
        }
    }
    //DISPOSING CODE
    private bool disposed = false;
    private void Dispose(bool disposing)
    {
        if (!this.disposed)
        {
            if (disposing)
            {
                dbContext.Dispose();
            }
```

```
            }
            this.disposed = true;
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

    }
}
```

OrderRepository Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Helpers;
using EShop.Models.DomainModel;
using System.Data;
using EShop.Models.DataAccess;
using System.Data.Objects;
using System.Data.Entity.Infrastructure;
using EShop.Models.Helpers;

namespace EShop.Models.Repositories
{
    public class OrderRepository : BaseRepository, IOrderRepository
    {
        public OrderRepository() : base(new EShopData()) { }

        #region CoreMethods

        public void Save(Order order, bool persistNow = true)
        {
            var orderInDB = dbContext.Orders.SingleOrDefault(o => o.OrderId == order.OrderId);
            if (orderInDB == null)
                dbContext.Orders.Add(order);

            if (persistNow)
                dbContext.SaveChanges();
        }

        public void Delete(int orderId, bool persistNow = true)
        {
            var order = this.GetById(orderId);
            if (order != null)
            {
                foreach (var product in order.Products)
                {
                    order.Products.Remove(product); //delete relationship
                    dbContext.OrderProducts.Remove(product); //delete from DB
                }
                dbContext.Orders.Remove(order);
            }
            if (persistNow)
                dbContext.SaveChanges();
        }

        public Order GetById(int orderId)
        {
            // eager-load product info
            var order = dbContext.Orders.Include("Products.Product").
```

```
            Include("Customer.Addresses").Include("Invoice")
            .SingleOrDefault(o => o.OrderId == orderId);

            return order;
        }

        public IList<Order> GetPagedAndSorted(SortAndPageCriteria sortAndPage, out int count)
        {
            var orders = dbContext.Orders.Include("Products.Product").Include("Customer");

            return base.GetPagedAndSorted(orders, sortAndPage, out count).ToList();
        }

        public void SaveForUnitOfWork()
        {
            dbContext.SaveChanges();
        }

        #endregion CoreMethods


        #region AdditionalMethods
        public IList<Order> GetByCustomerId(int customerId, out int count, SortAndPageCriteria
       sortAndPage = null)
        {
                        var orders = dbContext.Orders.Include("Products.Product").
                          Include("Customer").Where(o => o.CustomerId == customerId);

              count = orders.Count();

            if (sortAndPage != null)
                return base.GetPagedAndSorted(orders, sortAndPage).ToList();

            return orders.ToList();
        }

        public IList<Order> GetByDateRange(DateTime from, DateTime to, out int count,
       SortAndPageCriteria sortAndPage = null)
        {
                        var orders = dbContext.Orders.Include("Products.Product")
                                        .Include("Customer.Addresses").Include("Invoice")
                                        .Where(o => o.Created >= from && o.Created <=
                                        to).OrderBy(o => o.Created);
            count = orders.Count();

            if (sortAndPage != null)
                return base.GetPagedAndSorted(orders, sortAndPage).ToList();

            return orders.ToList();
        }

        #endregion AdditionalMethods

    }
}
```

OrdersController Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using EShop.Models.Repositories;
using EShop.Models.Helpers;
```

```
using EShop.Models.DomainModel;

namespace EShop.Areas.Admin.Controllers
{
    public class OrdersController : Controller
    {
        private readonly IOrderRepository _orderRepository;
        public OrdersController(IOrderRepository orderRepository)
        {
            _orderRepository = orderRepository;
        }
        public OrdersController()
            : this(new OrderRepository())
        { }

        public ActionResult Index(string timeScale, string sortBy = null, int page = 1)
        {
            DateTime from = DateTime.Now;
            DateTime to = DateTime.Now;
            int numberOfOrders = 0;
            TimeScale tScale = TimeScale.All;
            if (!String.IsNullOrEmpty(timeScale))
                Enum.TryParse<TimeScale>(timeScale, out tScale);

            DateTimeHelper.GetFromToDateTimes(ref from, ref to, tScale);
            sortBy = String.IsNullOrEmpty(sortBy) ? "created_desc" : sortBy;
            ViewBag.SortBy = sortBy;
            ViewBag.TimeScale = tScale.ToString();
            var sortAndPage = new SortAndPageCriteria(sortBy, ConfigHelper.PageSize, page);

            var orders = _orderRepository.GetByDateRange(from, to, out numberOfOrders, sortAndPage);
              ViewBag.Pager = new Pager(numberOfOrders, ConfigHelper.PageSize,
              Request.Url.AbsoluteUri);

            return View("Index", orders);
        }

        [HttpGet]
        public ActionResult Details(int orderId)
        {
            var order = _orderRepository.GetById(orderId);
            return View("Details", order);
        }
        [HttpPost]
        public ActionResult Details(int orderId, string orderStatus)
        {
            var order = _orderRepository.GetById(orderId);
            OrderStatusEnum orderStatusAsEnum = order.OrderStatusAsEnum;
            Enum.TryParse<OrderStatusEnum>(orderStatus, out orderStatusAsEnum);
            order.OrderStatusAsEnum = orderStatusAsEnum;
            _orderRepository.Save(order);

            return View("Details", order);
        }
        protected override void Dispose(bool disposing)
        {
            _orderRepository.Dispose();
            base.Dispose(disposing);
        }

    }
}
```