MAX Mapper Documentation

Table of Contents

[Introduction 4](#_Toc298946534)

[Getting Started 5](#_Toc298946535)

[The Business Domain Layer 5](#_Toc298946536)

[Defining the Service Contracts 7](#_Toc298946537)

[Creating the Service Layer 13](#_Toc298946538)

[What about Code First Entity Framework ? 21](#_Toc298946539)

[What about NHibernate or other object sources ? 22](#_Toc298946540)

[Mapping Tutorial 23](#_Toc298946541)

[Creating a Mapping 23](#_Toc298946542)

[Adding references 23](#_Toc298946543)

[Adding namespaces 23](#_Toc298946544)

[Mapping classes and enumerations 23](#_Toc298946545)

[Mapping scalar properties 23](#_Toc298946546)

[Mapping relational properties 23](#_Toc298946547)

[Mapping inheritance 23](#_Toc298946548)

[Flattening object graphs 23](#_Toc298946549)

[Flattening inheritance 23](#_Toc298946550)

[Unflattening object graphs 23](#_Toc298946551)

[Unflattening inheritance 23](#_Toc298946552)

[Using expressions 23](#_Toc298946553)

[Configuring reverse mapping 23](#_Toc298946554)

[Invoking the Mapper class 23](#_Toc298946555)

[Invoking the ReverseMapper class 23](#_Toc298946556)

[Mapping Reference 24](#_Toc298946557)

[Mapping element (root) 24](#_Toc298946558)

[Reference element 24](#_Toc298946559)

[Namespace element 25](#_Toc298946560)

[Type element 26](#_Toc298946561)

[KnownSubtype element 27](#_Toc298946562)

[Map element 28](#_Toc298946563)

[Attribute element 32](#_Toc298946564)

[Set element 32](#_Toc298946565)

[Generation Templates Reference 33](#_Toc298946566)

[Contract Layer Templates 33](#_Toc298946567)

[ContractsGenerator.tt 33](#_Toc298946568)

[ContractType.tt 33](#_Toc298946569)

[EnumType.tt 33](#_Toc298946570)

[ContractType\_ObjectOverrides.tt 34](#_Toc298946571)

[ContractType\_IEntityWithKey.tt 34](#_Toc298946572)

[ContractType\_IExtensibleDataObject.tt 34](#_Toc298946573)

[ContractModelValidation.tt 34](#_Toc298946574)

[Service Layer / Mapping Templates 34](#_Toc298946575)

[MapperGenerator.tt 34](#_Toc298946576)

[Mapper.tt 34](#_Toc298946577)

[ReverseMapper.tt 34](#_Toc298946578)

[Mapper\_GetSourcePropertyName.tt 34](#_Toc298946579)

[IdentifierBasedObjectSource.tt 34](#_Toc298946580)

[SelectProjections.tt 34](#_Toc298946581)

[Custom templates 35](#_Toc298946582)

[MasterTemplate.tt 35](#_Toc298946583)

[SubTemplate.tt 35](#_Toc298946584)

[Advanced Topics 36](#_Toc298946585)

[Understanding master/sub template code generation 36](#_Toc298946586)

[Understanding generated mapper code 37](#_Toc298946587)

[Extending and Customizing code generation 37](#_Toc298946588)

[Frequently Asked Questions 39](#_Toc298946589)

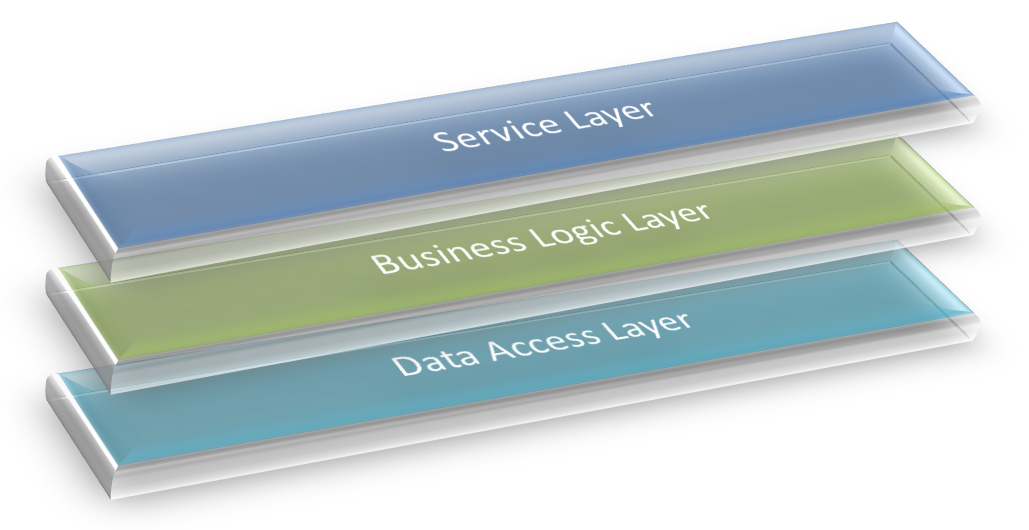
# Introduction

The Max Mapper Generator offers a solution for generating service contract types and back and forth mapper methods based on business entities and a mapping definition.

Service contract types, often referred to as data transfer objects (DTO’s) are a clean and technology independent way of defining service contracts that are stable and maintainable.

In a classical layered application, the Business Logic Layer contains business domain types such as Customer, Invoice, Product, Supplier, etc. defining both state and behavior that acts on that state.

The Data Access Layer persists domain types on a data store. Generic implementations of Data Access Layers include Entity Framework and Nhibernate. Often, the Data Access Layer is accessed by the Business Logic Layer by means of the Repository pattern.



The service layer is responsible of providing external access to the business logic through a well defined set of operations exposed through a specific protocol (WCF, Web services, .NET Remoting,...). Security, logging and other cross cutting concerns can hereby be applied.

The service layer should not directly expose the business domain types as their shape and behavior should be hidden from the service consumers. Instead, the service layer should map domain types to service contract types and back.

The Max Mapping Generation provides code generation for exact this purpose: creating service contract types and providing a two-way mapping between domain types and service contract types.

# Getting Started

In this tutorial we will develop an Ordering service that allows for consulting, placing and editing orders. We would like to have the following service operations:

* ListMyOrders()
* GetDetailedOrder(id) : anOrder
* SaveOrder(anOrder)

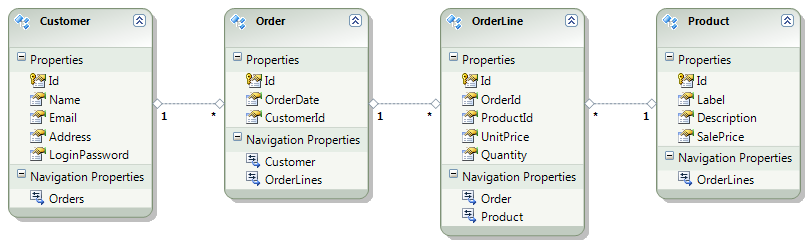
We will implement this service with an Entity Framework model, and generate contract types and forth and back mapping code.

## The Business Domain Layer

Before creating a service contract, we first need a business domain. Whether the domain layer was coded by hand or generated through Entity Framework or another code generation facility does not matter to the Max Mapper Generator, as long as you have a domain model.

(Note however that the way you created the domain model will influence on the options available to generate or use an out of the box implementation of a helper class for reverse mapping.)

Let’s consider for this tutorial the following simplified model of an order management system made with Entity Framework:



To this entity model we can add new partials of the domain classes to add derived properties and business behavior. For instance, we can add a TotalOrderPrice property to the Order entity by creating the following partial class:

using System.Linq;

namespace Ordering.BusinessLayer

{

    partial class Order

    {

        public decimal TotalOrderPrice

        {

            get { return OrderLines.Sum(line => line.Quantity \* line.UnitPrice); }

        }

    }

}

Summary

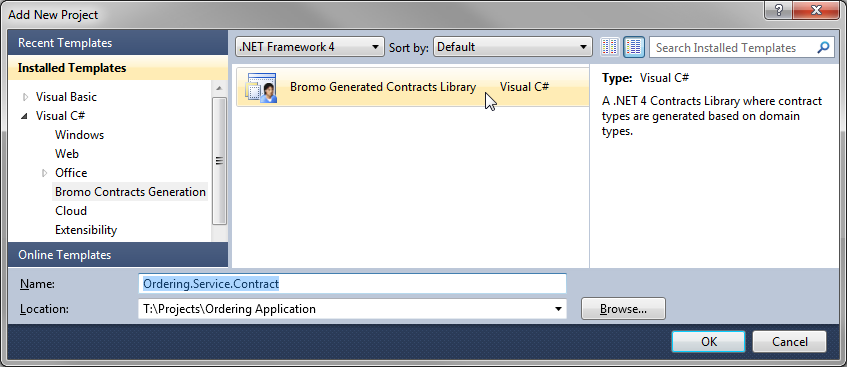
To summarize, we create the Business Domain Layer with the following steps:

1. Create a .NET Library project.
2. Add business domain classes, either by hand coding, using Entity Framework, or any other code generation system.

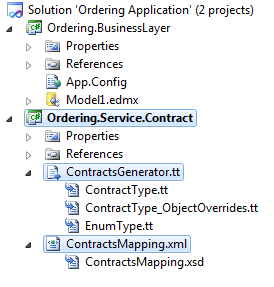
## Defining the Service Contracts

Next, we can define a service contract library, that’s a library project containing service contract types. This library does not need a reference to the business layer and the types contained in the contract library should be (close to) POCO’s.

We can add a Service Contracts project by adding a new project of type “Max Generated Contracts Library” to our solution:



The solution now includes an “Ordering.Service.Contract” library project that contains a ContractsGenerator template and a ContractsMapping XML file. There is also an XSD of the XML file.

The ContractsMapping.xml file is the place where the service contract types and their mapping is defined. This mapping consists of the definition of the contract types and how they map from/to business domain types.

The first step in defining the mapping, is to reference the business domain project(s) from the mapping XML file. This is done with the following line in the XML file:

  <reference project="Ordering.BusinessLayer"/>

Additional projects can be referenced, either by project name (if the project is in the current solution) or by assembly file.

Next, namespaces must be defined. At least one namespace should have the “Contract” alias, that is the namespace that will be used to generate contract types to. Additional namespaces can be added; using statements for these namespaces will be added in all generated files.

Finally, the contract types are to be defined. For instance, to define a *ProductITem* contract type, that would represent a Product in a list, we could write:

  <type name="ProductItem" source="Ordering.BusinessLayer.Product">

    <map property="Id" identifier="true"></map>

    <map property="Label"></map>

    <map property="Description"></map>

    <map property="SalePrice"></map>

  </type>

Hence, we define a type named ProductItem, that maps to the “Ordering.BusinessLayer.Product” domain layer type, and that takes the properties Id, Label, Description and SalePrice. The indication that Id is an identifier property is not required in all situations, but it’s a good practice to add it anyway.

The whole ContractsMapping.xml file now contains:

<?xml version="1.0" encoding="utf-8" ?>

<mapping namespace="urn:Ordering.Service.Contract"

         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

         xsi:noNamespaceSchemaLocation="ContractsMapping.xsd" >

  <!-- Add project references and namespaces to use here -->

  <reference project="Ordering.BusinessLayer"/>

  <!-- Namespace with alias "Contract" defines the contract types namespace -->

  <namespace name="Ordering.Service.Contract" alias="Contract" />

  <!-- Add namespaces to be imported in generated files -->

  <!-- Describe your mapping here -->

  <type name="ProductItem" source="Ordering.BusinessLayer.Product">

    <map property="Id" identifier="true"></map>

    <map property="Label"></map>

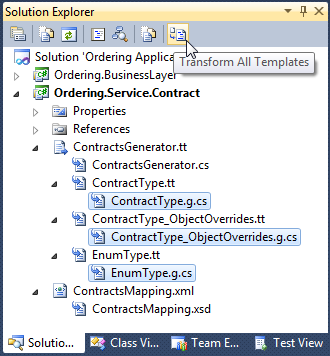
    <map property="Description"></map>

    <map property="SalePrice"></map>

  </type>

</mapping>

That’s all it takes to define a contract type. We can now generate the code by clicking the “Transform All Templates” button in the Solution Explorer:



As a result, several files will be generated: ContractType.g.cs, ContractType\_ObjectOverrides.g.cs and EnumType.g.cs. Take a time to look at the content of these files.

The whole set of contract types could include an OrderItem to show a list of orders, and CustomerItem, OrderDetail, OrderLineDetail and ProductItem for displaying and editing order details. The ContractsMapping.xml file then looks like:

<?xml version="1.0" encoding="utf-8" ?>

<mapping namespace="urn:Ordering.Service.Contract"

         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

         xsi:noNamespaceSchemaLocation="ContractsMapping.xsd" >

  <!-- Add project references and namespaces to use here -->

  <reference project="Ordering.BusinessLayer"/>

  <!-- Namespace with alias "Contract" defines the contract types namespace -->

  <namespace name="Ordering.Service.Contract" alias="Contract" />

  <!-- Add namespaces to be imported in generated files -->

  <!-- Describe your mapping here -->

  <type name="OrderItem" source="Ordering.BusinessLayer.Order">

    <map property="OrderNumber" source="Id" identifier="true"></map>

    <map property="OrderDate"></map>

    <map property="CustomerId"></map>

    <map property="CustomerName" source="Customer.Name"></map>

    <map property="TotalOrderPrice"></map>

    <map property="ProductIds" source="OrderLines.ProductId"></map>

    <map property="ProductLabels" source="OrderLines.Product.Label"></map>

  </type>

  <type name="CustomerItem" source="Ordering.BusinessLayer.Customer"

        reverseMapping="referenceOnly">

    <map property="Id" identifier="true"></map>

    <map property="Name"></map>

    <map property="Email"></map>

    <map property="Address"></map>

  </type>

  <type name="OrderDetail" source="Ordering.BusinessLayer.Order"

        reverseMapping="createAndUpdate">

    <map property="OrderNumber" source="Id" identifier="true"></map>

    <map property="OrderDate"></map>

    <map property="Customer" type="CustomerItem" conversion="map"></map>

    <map property="OrderLines" type="OrderLineDetail"  conversion="map"

         onRemove="delete"></map>

  </type>

  <type name="OrderLineDetail" source="Ordering.BusinessLayer.OrderLine"

        reverseMapping="createAndUpdate">

    <map property="Id" identifier="true"></map>

    <map property="Product" type="ProductItem" conversion="map"></map>

    <map property="UnitPrice"></map>

    <map property="Quantity"></map>

  </type>

  <type name="ProductItem" source="Ordering.BusinessLayer.Product"

        reverseMapping="referenceOnly">

    <map property="Id" identifier="true"></map>

    <map property="Label"></map>

    <map property="Description"></map>

    <map property="SalePrice"></map>

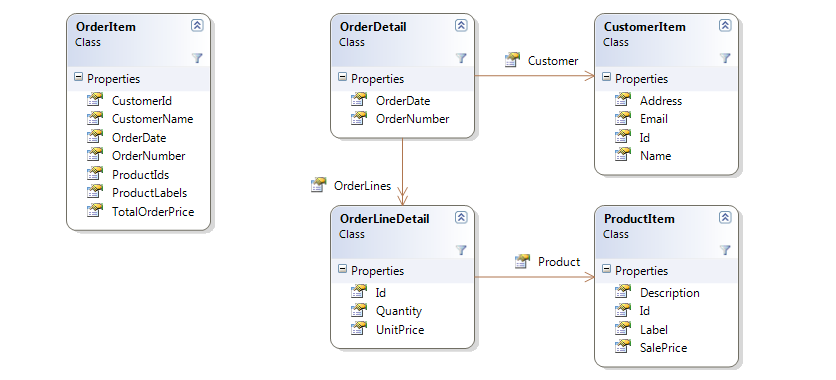
  </type>

</mapping>

This mapping contains a few features that are worth some explanation. The referred mappings have been highlighted in the XML:

* The OrderItem.OrderNumber property has a source of “Id”, this means that on the contract type, there should be an “OrderNumber” property (instead of an “Id”), but that that property maps to the “Id” property in the domain model.
* The OrderItem.CustomerName property maps to the Name of the Customer of the order, hence it has a source of “Customer.Name”. This is known as *flattening*.
* Similar for the OrderItem.ProductIds which will result in a list of integers. Note that we do not need to do something special to navigate the “OrderLines” collection in the source, the source expression “OrderLines.ProductId” will be translated into a correct C# expression (*i.e: OrderLines.Select(o => o.Id)*) by the mapper code generation as we will see later.
* The CustomerItem class has an attribute reverseMapping set to referenceOnly. This means that CustomerItem contract objects can be used for referencing particular customers, but not for editing it’s properties or creating new customers. This is also a topic we’ll cover later.
* The CustomerItem class has no mapping for the LoginPassword property on the Customer domain object. Obviously, we do not want to expose that property!
* The OrderDetail (and OrderLineDetail) class has the reverseMapping attribute set to createAndUpdate. This means that an OrderDetail can be used for creating new orders and editing new ones.
* The OrderDetail.Customer property maps to the Customer property of a business domain Customer. Since this property is itself of type a business domain type, it needs to be translated (mapped) to a contract type. The request to do this translation comes from the conversion="map" attribute, and the type attribute indicates to which contract type the Customer domain object should be translated (here, CustomerItem).
* Similar for the OrderDetail.OrderLines property. Note however that OrderLines is a collection and that the type attribute is used to tell the collection member type.
* In addition, the OrderDetail.OrderLines property has the onRemove attribute set to “delete”, instructing to delete orderlines that have been removed from their order.

If we regenerate the code and create a class diagram, we see the following service contract types defined:



It is interesting to note that an Order business domain type can be translated into 2 different types in the contract model: OrderItem or OrderDetail. The last one represents a whole objectgraph of the order while the first one has ‘flattened’ some properties into a single type. This capability of having differently shaped representations of the same business domain types on your service contract is important as it is the key to constructing secure, maintainable and performant services.

The generated contract types are partial and have several partial methods to provide you maximum flexibility in extending and customizing the generated contract types.

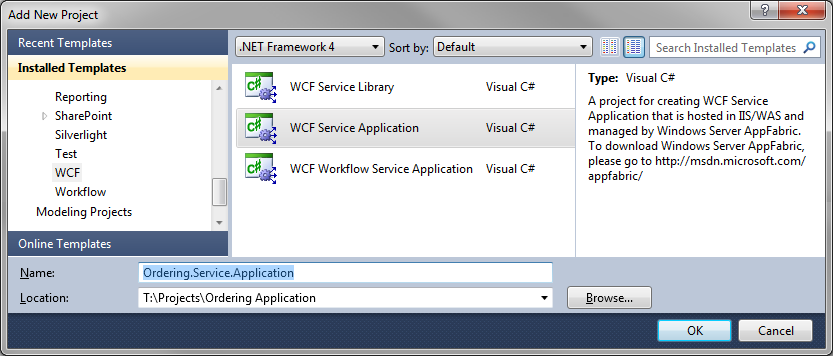
Summary

To summarize, we create a service contract with the following steps:

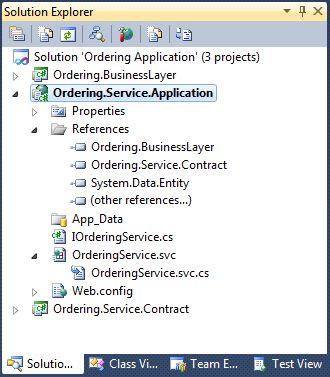
1. Add a “Max Generated Contracts Library” project to your solution.
2. Edit the ContractsMapping.xml to define the service contract types and their mapping to business domain types.
3. Run the “Transform All Templates” tool of Visual Studio to generate or update the contract types.

## Creating the Service Layer

Now that the Service Contract library is ready, we can create the service implementation. In this tutorial we’ll choose for a “WCF Service Application”:



To the service project we add references to both the domain layer and the service contract library. Since we use Entity Framework in our business layer, we also need to add a reference to the System.Data.Entity assembly. After renaming the interface type and service type our project has now the following outline (note that only added references are shown):



Also don’t forget to copy the Entity Framework connectionstring from the App.config file in the Ordering.BusinessLayer to the Web.config of the service application, as it is the service application that will form the entry assembly towards the business layer.

We can now define the service contract IOrderingService using the contract types as arguments and return types of our service operations:

using System.Collections.Generic;

using System.ServiceModel;

using Ordering.Service.Contract;

namespace Ordering.Service.Application

{

    [ServiceContract]

    public interface IOrderingService

    {

        [OperationContract]

        IList<CustomerItem> ListAllCustomers();

        [OperationContract]

        IList<OrderItem> ListOrdersOfCustomer(int customerId);

        [OperationContract]

        OrderDetail GetDetailedOrder(int orderNumber);

        [OperationContract]

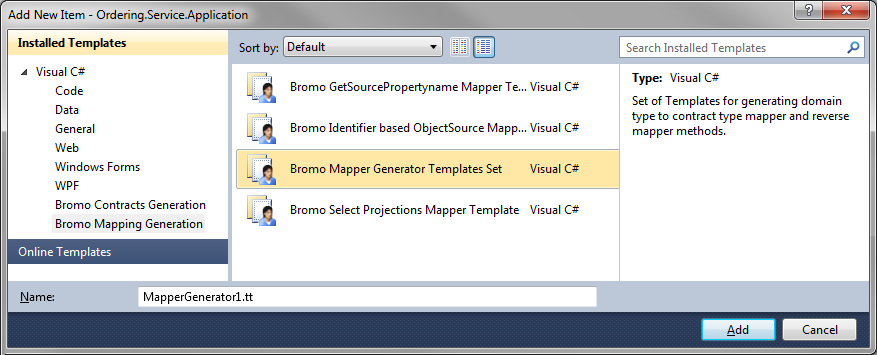
        int SaveOrder(OrderDetail newOrEditedOrder);

    }

}

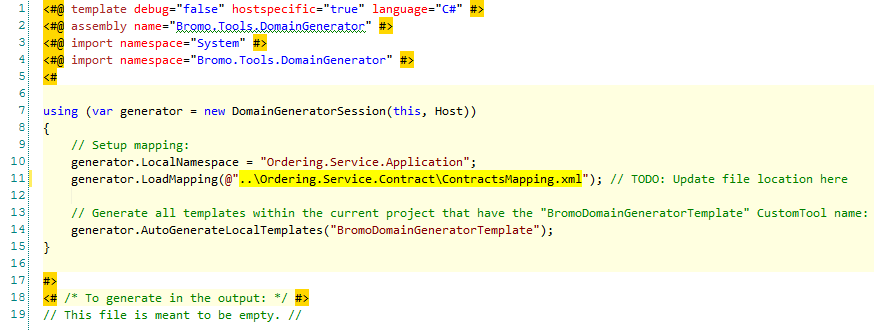
When it comes to implementing this service contract, the whole issue of mapping business domain types to contract types is showing up. What will be the implementation of the ListAllCustomers operation ? Simply “return new OrderingContext().Customers;” is not enough, we need to translate the resulting object, of type Ordering.BusinessLayer.Customer, to the contract type Ordering.Service.Contract.CustomerItem.

This translation or mapping can be generated by adding a “Max Mapper Generator Tempates Set” to the service application project:

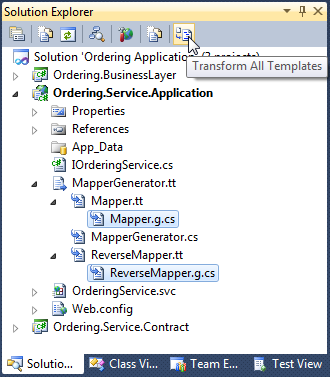


(Don’t bother giving a name to the item, it will always be included with the name “MapperGenerator.tt”.)

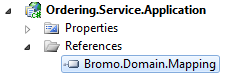
Before running the generator, we need to update one line of code in it: we need to tell it where to find the ContractsMapping.xml file. For this, update the file path ad argument of the LoadMapping method call in the MapperGenerator.tt file. Relative paths are allowed, for instance:



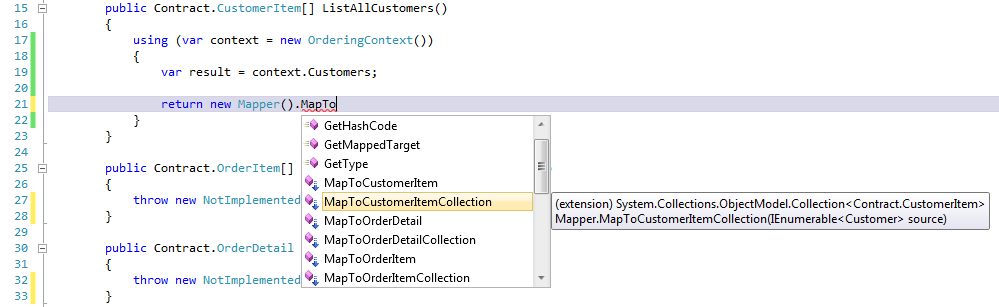
Then, after pressing the “Transform All Templates” on top of the Solution Explorer, we should see the files “Mapper.g.cs” and “ReverseMapper.g.cs” being generated:



You should also note that the project has now an added reference to the “Max.Domain.Mapping” assembly:



This assembly defines two classes for mapping: Mapper and ReverseMapper. The Mapper class is used to translate business domain types into service contract types. This is done in combination with the code generated by the Max Mapper Generator templates which generate extension methods of both the Mapper and the ReverseMapper class:



We can now implement the whole service, with exception of the SaveOrder operation, using the Mapper extension methods to translate domain types to contract types. The whole service implementation so far is the following:

using System;

using System.Collections.Generic;

using System.Linq;

using Max.Domain.Mapping;

using Ordering.BusinessLayer;

namespace Ordering.Service.Application

{

    public class OrderingService : IOrderingService

    {

        public IList<Contract.CustomerItem> ListAllCustomers()

        {

            using (var context = new OrderingContext())

            {

                var result = context.Customers;

                return new Mapper().MapToCustomerItemCollection(result);

            }

        }

        public IList<Contract.OrderItem> ListOrdersOfCustomer(int customerId)

        {

            using (var context = new OrderingContext())

            {

                var result = context.Orders.Where(o => o.CustomerId==customerId);

                return new Mapper().MapToOrderItemCollection(result);

            }

        }

        public Contract.OrderDetail GetDetailedOrder(int orderNumber)

        {

            using (var context = new OrderingContext())

            {

                var result = context.Orders.SingleOrDefault(o=>o.Id==orderNumber);

                return new Mapper().MapToOrderDetail(result);

            }

        }

        public int SaveOrder(Contract.OrderDetail newOrEditedOrder)

        {

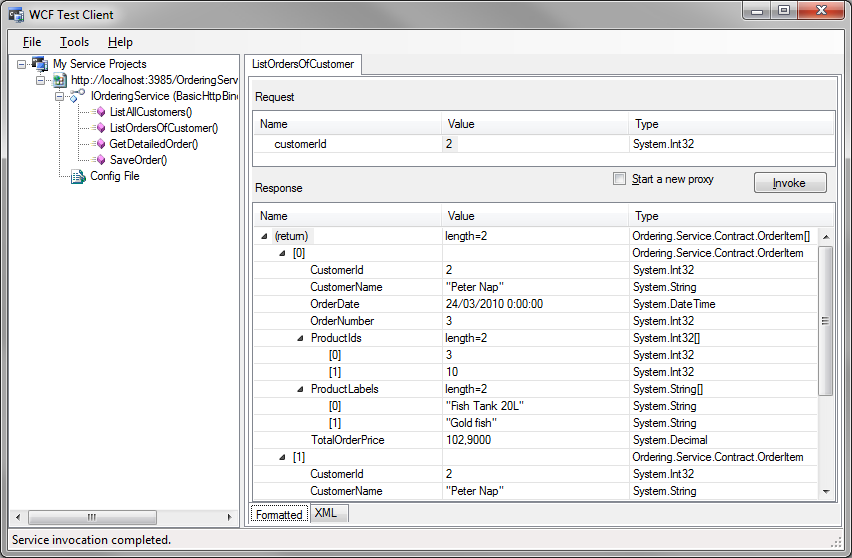
            throw new NotImplementedException();

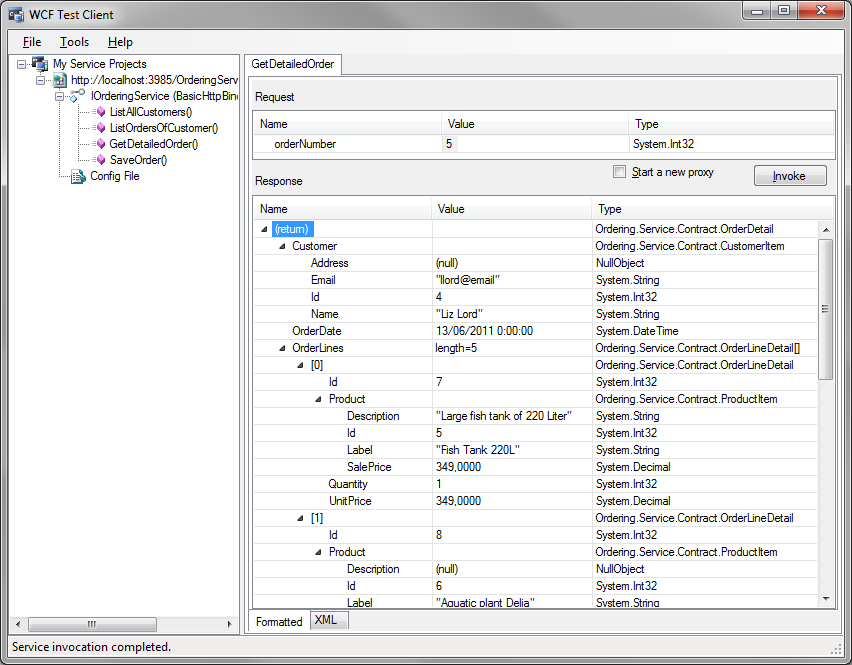
        }

    }

}

If we have some data available in the database, we can now test this service using for instance the WCF Test Client that comes with Visual Studio:





The SaveOrder operation is different from the other operations because it requires the ability to translate an object of a service contract type into an object of the business domain layer, an action I called *reverse mapping*.

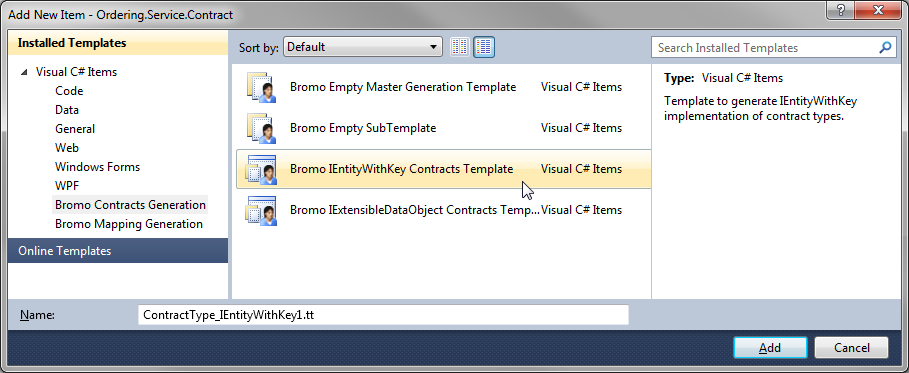
Do do reverse mapping, we can use the ReverseMapper class defined in the Max.Domain.Mapping assembly combined with it’s extension methods generated by the ReverseMapping.tt template, similar to using the Mapper class. However, the ReverseMapper needs a connection with the object source (whether it is Entity Framework, NHibernate, a repository pattern implementation or any other kind of object source). For this, its constructor expects an argument of type IObjectSource. Question is, where do we get an implementation of this interface ?

If you are using your own object source, you’ll have to provide an implementation yourself. When using Entity Framework, 2 options are provided out of the box. The first one is to use the EntityModelObjectSource class defined in the Max.Domain.Mapping.Entity assembly. This implementation requires both our business layer types and contract types to implement IEntityWithKey, an interface of Entity Framework.

Provided that we have defined the entities using an EDMX file (an ADO.NET Entity Data Model file), which is the case here, or are referencing a compiled assembly that had such a file, we can use the “Max IEntityWithKey Contracts Template” to have our contract types implement IEntityWithKey. The business layer types already implement IEntityWithKey since we used the default EDMX code generation.

We proceed with the following steps to have our contract types implement IEntityWithKey:

1. We add the “Max IEntityWithKey Contracts Template” to the Ordering.Service.**Contract** project:



2. We now also need to add a reference to the System.Data.Entity assembly in the contract project.

3. We regenerate the code with the “Transform All Templates” command in Visual Studio.

A file “ContractType\_IEntityWithKey.g.cs” has now been generated. Note that the implementation is such, that the serialized form of contract objects has not changed. Therefore this change should not impact compatiblity.

To use the EntityModelObjectSource object source with the ReverseMapper, we still need to add a reference to the Max.Domain.Mapping.Entity assembly in the Ordering.Service.**Application** project. This assembly can be found in the .NET assemblies of the Add Reference dialog box:

|  |  |
| --- | --- |
|  |  |

We are now able to implement the SaveOrder operation:

       [OperationBehavior(TransactionScopeRequired = true)]

       public int SaveOrder(Contract.OrderDetail newOrEditedOrder)

       {

           using (var context = new OrderingContext())

           {

               var order = new ReverseMapper(new EntityModelObjectSource(context))

                   .AllowCreatingAndUpdating<Order>()

                   .AllowCreatingAndUpdating<OrderLine>()

                   .MapFromOrderDetail(newOrEditedOrder);

               context.SaveChanges();

               return order.Id;

           }

       }

The ReverseMapper takes an IObjectSource constructor argument. Furthermore, we have to instruct the ReverseMapper which business layer types are allowed to be created or updated. This is specific to this service operation as it defines part of the semantics of this service operation: the SaveOrder operation will create or update orders and orderlines, but will not touch customer or product objects!

Finally, we use one of the generated MapFrom methods to map a contract type object into a domain type object. New objects are automatically added to the Entity Framework context, and so, unless we want to do something more such as business validation, it is sufficient to call the SaveChanges() method of the Entity Framework context to have the order persisted.

Summary

To summarize, we create a service application with the following steps:

1. Add the service project to your application.
2. Add the “Max Mapper Generator Templates Set” to the service project, and update the MapperGenerator.tt file to point to the right ContractsMapping.xml file.
3. If you plan to use reverse mapping, ensure you have an implementation of IObjectSource, for instance by having your contract types implement IEntityWithKey and use the EntityModelObjectSource class.
4. Run the “Transform All Templates” tool of Visual Studio to generate or update the mapping code.
5. Implement your service using the Mapper and ReverseMapper classes to map forth and back domain types to contract types.

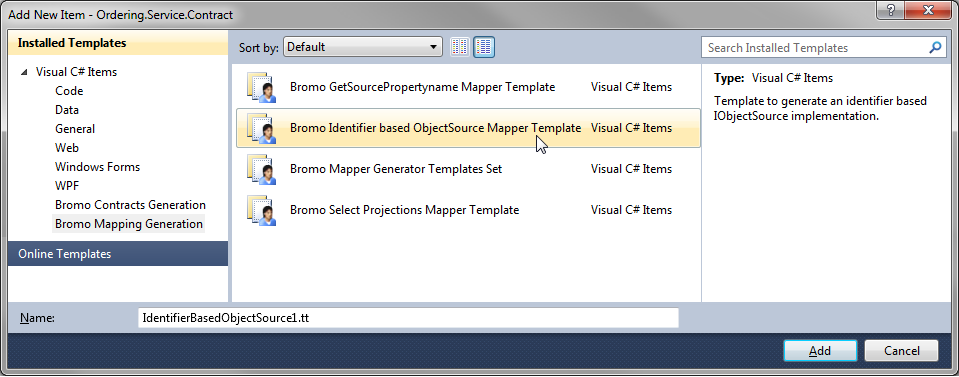
## What about Code First Entity Framework ?

In this tutorial we have used an Entity Framework Database First or Model First approach which results in having an EDMX file. The EDMX file was important as it allowed us to implement IEntityWithKey on our contracts, which in turn was required to use the EntityModelObjectSource class with the ReverseMapper.

When following a Code First approach with Entity Framework 4.1, we don’t get an EDMX file and therefore can’t use this option.

For Entity Framework Code First, you can follow these steps:

1. To the service implementation project (Ordering.Service.Application in the sample), add the “Max Identifier based ObjectSource Mapper Template”:



This template generates an ObjectSource implementation that does not require your contract types to be IEntityWithKey.

2. Add a reference to the “EntityFramework” assembly that comes with Entity Framework 4.1 to the service implementation project.

The SaveOrder method could now be written using the generated IdentifierBasedObjectSource class:

   [OperationBehavior(TransactionScopeRequired = true)]

   public void SaveOrder(Contract.OrderDetail newOrEditedOrder)

   {

       using (var context = new CodedOrderingContext())

       {

           var order = new ReverseMapper(new IdentifierBasedObjectSource(context))

               .AllowCreatingAndUpdating<Order>()

               .AllowCreatingAndUpdating<OrderLine>()

               .MapFromOrderDetail(newOrEditedOrder);

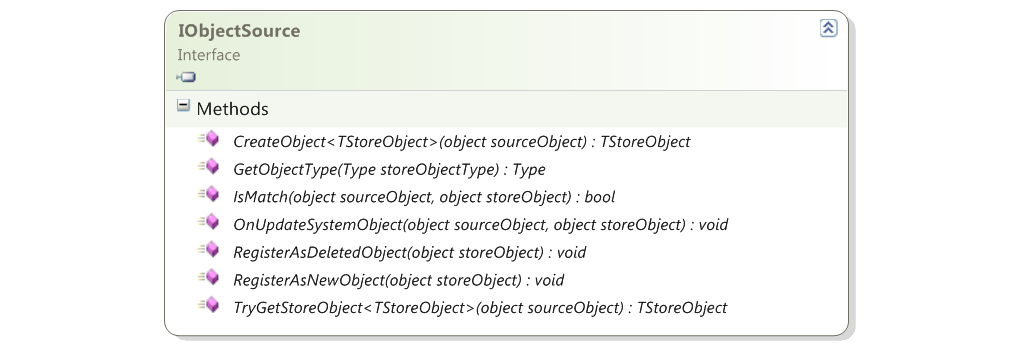
           context.SaveChanges();

       }

   }

## What about NHibernate or other object sources ?

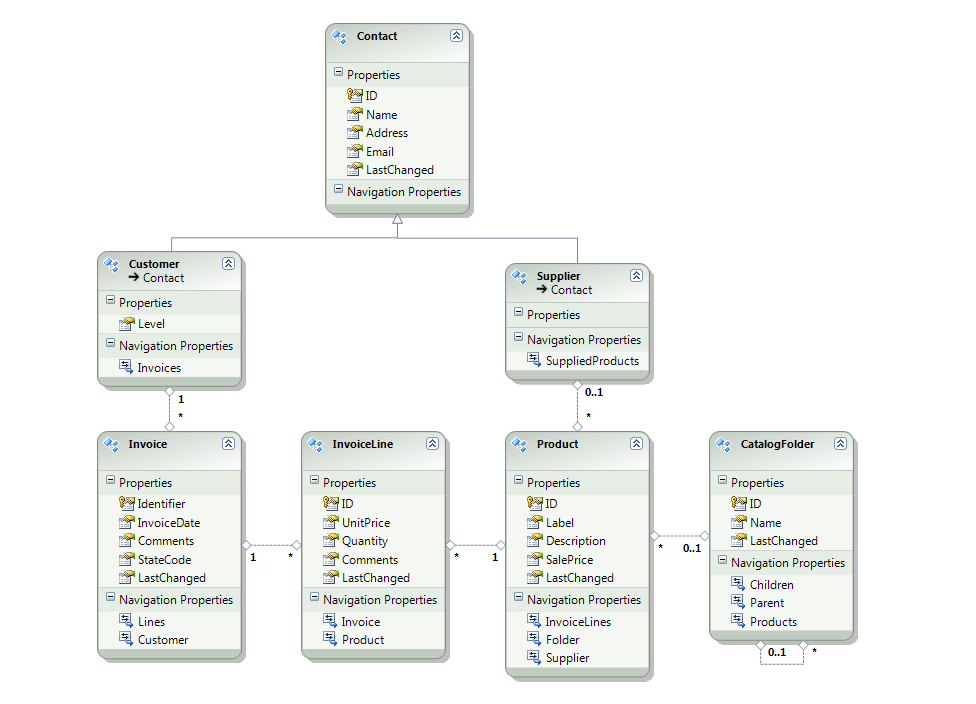
Currently, no out of the box solution is provided for other object sources than Entity Framework, but you can provide yourself an implementation of IObjectSource by implementing the following interface:



# Mapping Tutorial

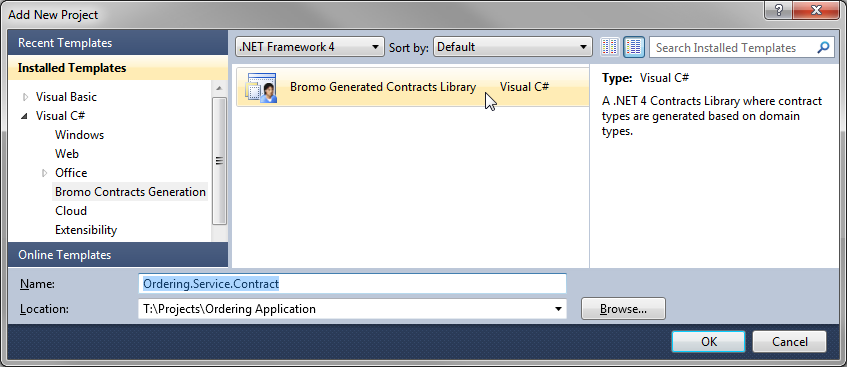
This chapter explains how to write mapping definitions and use the Mapper classes within service layers of applications.

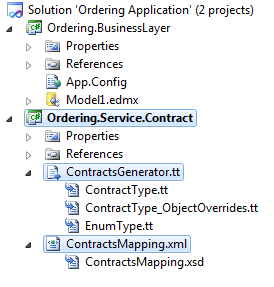
Throughout this chapter, we will use the following business domain for most samples. The model is similar to the one in the *Getting Started* chapter except it is a little more complex and contains inheritance.



## Creating a Mapping

Basically, a mapping, or a mapping definition file, is created as part of a “Max Generated Contracts Library”. To create a mapping definition file, you need to create a contracts library project, as only one mapping definition and associated set of code generation templates can live in a project.



The generated project then contains a ContractsMapping.xml file with the mapping definition, as well as a master code generation template (ContractsGenerator.tt) and a few sub code generation templates (ContractType.tt,…).

You can now start defining the mapping in the ContractsMapping.xml file. Note that the project also contains a ContractsMapping.xsd file which will help you defining the mapping as it provides auto completion and context sensitive tooltips while editing the ContractsMapping.xml mapping definition.

## Adding references

The first step in defining the mapping, is defining the mapping source references.

This is done by adding <reference> elements in the mapping definition. Those reference elements can take two forms, depending on whether you are referencing another project within the same solution, in which case you refer to the name of the project with the project attribute:

<reference project="Invoicing.BusinessLayer"/>

Or whether you are referencing a precompiled assembly, in which case you refer to the assembly filename with the path attribute:

<reference path="..\Invoicing.BusinessLayer\bin\debug\Invoicing.BusinessLayer.dll"/>

References must be added using the reference element to all assemblies/projects containing types that are used from within the source domain model. For instance, if the source model contains a type Account with a property of type Money, the assemblies defining the types Account and Money must be referenced in the mapping XML file.

Note that these references in the mapping definition do not replace regular project references in Visual Studio.NET. While the contracts project will typically not need references to the source domain types, the project containing mapping code will.

## Adding namespaces

Next, namespace definitions are to be added to the mapping definition.

The contract types are generated in the namespace with alias “Contract”, and therefore, there must be a namespace with this alias. Mapper types are defined in the namespace with alias “Mapper”, or in the default namespace for the mapper project.

Additional namespaces are added as using lines in the generated code.

The following namespace should not be included as they are already included by default:

* System
* System.Collections.Generic
* System.Collections.ObjectModel
* System.Linq
* System.Runtime.Serialization
* System.ComponentModel.DataAnnotations

I.e:

<namespace name="Invoicing.Service.Contract" alias="Contract" />

<namespace name="System.ComponentModel" />

<namespace name="System.Globalization" />

Whenever generated code does not compile because a using is missing, adding the namespace declaration in the mapping definition should solve the problem.

## Mapping classes and enumerations

Both classes and enumerations are mapped using the <type> element. The mapping generator will detect that the source type is an enumeration or a class and will apply the correct code generation template. When mapping enumerations, the type element can have set and attribute elements, but should not have map or knownSubtype elements.

The mapping definition of a type mentions the name of the contract type to generate, and the source type’s full name (namespace + type name).

The mapping definition for an enumeration could look like:

  <type name="InvoiceStatus" source="Invoicing.BusinessLayer.InvoiceStatus">

  </type>

Important to note is that if the source type is in a project in the same solution, and that project is referenced by project, then the source enumeration must explicitly define the values of its members. This is anyhow a good practice, but becomes mandatory for source enumeration types. Therefore, the following type is not good:

    public enum InvoiceStatus

    {

        Open,

        Submitted,

        Handled,

        Paid,

        Closed,

    }

Instead, it should be:

    public enum InvoiceStatus

    {

        Open = 0,

        Submitted = 1,

        Handled = 2,

        Paid = 3,

        Closed = 4,

    }

The generated enumeration type will then look like:

    /// <summary>

    /// InvoiceStatus enumeration type.

    /// </summary>

    [DataContract(), Serializable()]

    public enum InvoiceStatus

    {

        [EnumMember] Open = 0,

        [EnumMember] Submitted = 1,

        [EnumMember] Handled = 2,

        [EnumMember] Paid = 3,

        [EnumMember] Closed = 4,

    }

The mapping definition of a class is similar to that of an enumeration, except it might have more options, and will usually also have map elements to define the mapping of the class’s properties.

A basic mapping definition of class could be:

  <type name="ProductItem" source="Invoicing.BusinessLayer.Product">

    <map property="ID" identifier="true"></map>

    <map property="Label"></map>

    <map property="Description"></map>

    <map property="SalePrice"></map>

  </type>

## Mapping scalar properties

The mapping of properties is defined with the <map> element of <type>. For scalar properties it is often sufficient to name the source property name or path, and the target property name. For instance:

  <type name="ProductItem" source="Invoicing.BusinessLayer.Product">

    <map property="Label" source="Label"></map>

  </type>

When both the source and the target are identical, the source can be left out, resulting in the minimal map element:

  <type name="ProductItem" source="Invoicing.BusinessLayer.Product">

    <map property="Label"></map>

  </type>

By default, the type of the mapped property will be the same as the type of the source property, which, for scalar properties is often wanted.

When types should be different, the target type must be given together with a chosen conversion method, which for scalar properties should be *cast*, *convert* or *custom*.

Cast means a regular type cast expression should be used. This is for instance valid when converting int to long, or int to int?. Convert means the System.Convert.ChangeType() method will be used which provides for a default and extensible conversion mechanism and can convert strings to numbers etc. Custom means you will provide code to perform the conversion. When custom conversion is chosen, the generated mapper code will call methods that you have to implement.

In the following example, the SalePrice property is casted from decimal to double:

  <type name="ProductItem" source="Invoicing.BusinessLayer.Product">

    <map property="SalePrice" type="double" conversion="cast"></map>

  </type>

## Mapping identifier properties

The mapping of identifier properties is identical to the mapping of other scalar properties, except for two points:

* The <map> element should have the identifier attribute set to true.
* The order of the properties is important, the set of identifier properties of a type must be ordered according to the database key order.

For instance, if InvoiceLine were to have no own Id identifier but a combined key:

  <type name="InvoiceLineDetail" source="Invoicing.BusinessLayer.InvoiceLine">

    <map property="InvoiceId" identifier="true"></map>

    <map property="LineNo" identifier="true"></map>

  </type>

## Mapping relational properties

Relational properties are properties that related business domain objects together. This means that both the type holding the property, and the type of the property must be mapped. As a consequence, the type of the property will always be different in the business domain model from the generated contract model, and thus type and conversion attributes must be given. However, a special conversion value – *map* – can be used to indicate conversion should be done by object mapping.

The following illustrates the mapping of the relational property that connects an Invoice to its Customer:

  <type name="CustomerItem" source="Invoicing.BusinessLayer.Customer">

    <map property="ID" identifier="true"></map>

    <map property="Name"></map>

  </type>

  <type name="InvoiceDetail" source="Invoicing.BusinessLayer.Invoice">

    <map property="ID" identifier="true"></map>

    <map property="Customer" type="CustomerItem" conversion="map"></map>

  </type>

When mapping (relational) collection properties, the type attribute of the <map> element takes the collection member type. (The mapper generator auto-detects whether a source property is a collection or not.) The relation between Invoice and InvoiceLine is an example of a collection:

  <type name="InvoiceDetail" source="Invoicing.BusinessLayer.Invoice">

    <map property="ID" identifier="true"></map>

    <map property="Customer" type="CustomerItem" conversion="map"></map>

    <map property="Lines" type="InvoiceLineDetail" conversion="map"></map>

  </type>

  <type name="InvoiceLineDetail" source="Invoicing.BusinessLayer.InvoiceLine">

    <map property="ID" identifier="true"></map>

  </type>

Note that the type name, when used with map conversion, is just the type name without namespace. The name is expected to match a <type> element in the same mapping definition (although that is not a strict requirement).

When defining the contract layer, great care must be taken to control recursive properties, especially when the domain layer is implemented with an ORM supporting lazy loading. Suppose the CustomerItem type of the above example would also have the (reverse) property “Invoices”. Querying one invoice the results in retrieving its customer but then also all the invoices of that customer. The object graph returned is then much larger than required which can cause severe performance and load penalties.

In some cases, recursion is not such a problem. For instance, if the above InvoiceLineDetail had an “Invoice” property, this would probably not be problematic since invoice lines are always returned in the context of their invoice. But then again, the property might be of low use while still causing some overhead.

## Flattening object graphs

When we do not which to expose the complex internal domain object model but want to expose a simplified view, object graph flattening is the most natural solution.

Object graph flattening means we will bring all properties that are dispersed over several related objects into one single root object.

When defining contract mappings, object graph flattening is done by setting the source of the property maps. In the following sample, the InvoiceItem does not expose its relation with the Customer type, but has flattened the structure by adding properties CustomerID, CustomerName, CustomerAddress and CustomerEmail on the InvoiceItem contract type:

  <type name="InvoiceItem" source="Invoicing.BusinessLayer.Invoice">

    <map property="Identifier" identifier="true"></map>

    <map property="CustomerID" source="Customer.ID"></map>

    <map property="CustomerName" source="Customer.Name"></map>

    <map property="CustomerAddress" source="Customer.Address"></map>

    <map property="CustomerEmail" source="Customer.Email"></map>

    <map property="InvoiceDate"></map>

    <map property="StateCode"></map>

  </type>

The flattening trick here consists of setting a source property that navigates through the object graph to retrieve the right property. Most often, the source path will only navigate singular properties, but it is also possible to navigate collection properties, as in this example:

  <map property="ProductLabels" source="Lines.Product.Label"></map>

In all cases, the mapper will gracefully handle null values in the path: if an invoice has no customer, the CustomerID, CustomerName, CustomerAddress and CustomerEmail will simply have their types default value (0 for the numerical ID, null for the string values). It would also have been possible to declare the CustomerID type as nullable:

  <map property="CustomerID" source="Customer.ID" type="long?"></map>

The source property takes a property path in which it is also allowed to include conditions between square brackets, as in:

  <map property="ProductLabels"

source="Lines.Product[p => p.ID &gt; 0].Label"></map>

(The “&gt;” translates to a GreaterThan (>) sign, “&lt;” to LessThan (<) sign in XML.)

## Unflattening object graphs

The reverse operation, unflattening the object graph, is also possible, although it requires a little of hand coding. The following mapping example will introduce a new ContactInformation property with the contact information of contacts:

  <type name="Customer" source="Invoicing.BusinessLayer.Customer">

    <map property="ID" identifier="true"></map>

    <map property="Name"></map>

    <map property="ContactInformation.Address" source="Address"></map>

    <map property="ContactInformation.Email" source="Email"></map>

  </type>

When generating the contracts, in fact, those two properties will simply be ignored, because the generator cannot know the type of ContactInformation, and whether or not this type is a class that must also be generated.

It is therefore up to you to implement a partial class of Customer which has a ContactInformation property. In addition, you should make sure that property is never null, for instance by initiating it at construction time. So basically, you need to add a partial class of Customer in the contracts project, with code similar to the following:

namespace Invoicing.Service.Contract

{

    partial class Customer

    {

        partial void OnInstanceCreated()

        {

            this.ContactInformation = new ContactInformation();

        }

        [DataMember()]

        public ContactInformation ContactInformation { get; private set; }

    }

    [DataContract, Serializable]

    public class ContactInformation

    {

        [DataMember()]

        public string Address { get; set; }

        [DataMember()]

        public string Email { get; set; }

    }

}

## Mapping inheritance

In the invoicing model, Customer and Supplier inherit from a common Contact type. Several options are available towards mapping those business types into contract types. If we want to keep the inheritance relationship and create a similar inheritance hierarchy in the contracts project, we merely define the 3 types, the property mappings of the properties declared on their source type, and indicate the subtypes have a base type. Eventually, base types can be marked abstract.

  <type name="ContactItem" source="Invoicing.BusinessLayer.Contact"

 abstract="true">

    <map property="ID" identifier="true"></map>

    <map property="Name"></map>

    <map property="Email"></map>

  </type>

  <type name="CustomerItem" source="Invoicing.BusinessLayer.Customer"

baseType="ContactItem">

    <map property="Level" conversion="cast" type="CustomerLevel?"></map>

  </type>

  <type name="SupplierItem" source="Invoicing.BusinessLayer.Supplier"

baseType="ContactItem">

  </type>

This mapping definition will generate three contract types and their mapping code as expected. However, when using the mapper method MapToContactItem() with as argument for instance a Customer entity, the mapper will not try to create a CustomerItem, but will try to create, as asked, a ContactItem object. Since ContactItem is abstract, this will ultimately result in a runtime mapper exception.

Promotion to a subtype is only done for subtypes that are explicitely declared as known subtypes using the <knownSubtype> element on the base type.

  <type name="ContactItem" source="Invoicing.BusinessLayer.Contact"

 abstract="true">

    <knownSubtype>CustomerItem</knownSubtype>

    <knownSubtype>SupplierItem</knownSubtype>

    <map property="ID" identifier="true"></map>

    <map property="Name"></map>

    <map property="Email"></map>

  </type>

The <knownSubtype> element has two effects:

* When mapping objects, promotion to the right subtype is performed. For instance, the mapper MapToContactItem will now return an instance of CustomerItem or SupplierItem.
* The contract type ContactItem also gets [KnownType] attributes needed to perform the same promotion through WCF.

## Flattening inheritance

An alternative mapping of class hierarchies is to flatten the hierarchies. If for instance the Contact base type should be seen as an internal design choice of the business domain layer and should not be exposed through contracts, the following mapping could be defined:

  <type name="CustomerItem" source="Invoicing.BusinessLayer.Customer">

    <map property="ID" identifier="true"></map>

    <map property="Name"></map>

    <map property="Email"></map>

    <map property="Level" conversion="cast" type="CustomerLevel?"></map>

  </type>

  <type name="SupplierItem" source="Invoicing.BusinessLayer.Supplier">

    <map property="ID" identifier="true"></map>

    <map property="Name"></map>

    <map property="Email"></map>

  </type>  
  
With this mapping definition, there is also no need to define known subtypes as there is no (abstract) base type defined.

## Unflattening inheritance

While this is a less commonly used feature, it is also possible to expose an inheritance hierarchy where none exists in the business domain layer.

Suppose we would like to expose two product types: PricedProduct and a NonPricedProduct, based on the same business domain type Product, and have a BaseProduct base type. The following mapping definition does the trick:

  <type name="BaseProduct" source="Invoicing.BusinessLayer.Product"

abstract="true">

    <knownSubtype>PricedProduct</knownSubtype>

    <knownSubtype>NonpricedProduct</knownSubtype>

    <map property="ID" identifier="true"></map>

    <map property="Label"></map>

    <map property="Description"></map>

  </type>

  <type name="PricedProduct" source="Invoicing.BusinessLayer.Product"

baseType="BaseProduct" typeCondition="source.SalePrice.HasValue">

    <map property="SalePrice"></map>

  </type>

  <type name="NonpricedProduct" source="Invoicing.BusinessLayer.Product"

 baseType="BaseProduct" typeCondition="!source.SalePrice.HasValue">

  </type>

Basically, we define the 3 types with the same business domain type source Product, but with the baseType attribute to design the hierarchy, and a typeCondition on each base type that is mutually exclusive and that will allow the mapper to determine which object type to return.

It is now possible to use the mappers MapToBaseProduct() to obtain either a PricedProduct or a NonpricedProduct.

## Using expressions

## Configuring reverse mapping

## Mapping to multiple contract types

It is possible to map business types to different kinds of contract types. And this is also very useful. Consider the contract types as ‘views’ of the business types, that may have a different look and feel from their originating business type, and may evolve differently.

Take for instance the Invoice type. Depending on whether we want to show a list of invoices, refer to invoices or allow creation and editing of invoices, we might want to choose different representations of an invoice on the client side of our application.

Typically, we would have a non-detailed and non-editable representation used for list rendering and referencing, and a detailed editable representation for creation and editing, resulting in two contract types InvoiceItem and InvoiceDetail:

  <type name="InvoiceItem" source="Invoicing.BusinessLayer.Invoice"

reverseMapping="referenceOnly">

    <map property="ID" identifier="true"></map>

    <map property="CustomerID" source="Customer.ID"></map>

    <map property="CustomerName" source="Customer.Name"></map>

  </type>

  <type name="InvoiceDetail" source="Invoicing.BusinessLayer.Invoice"

reverseMapping="updateAndCreate">

    <map property="ID" identifier="true"></map>

    <map property="Customer" type="CustomerItem" conversion="map"></map>

    <map property="Lines" type="InvoiceLineDetail" conversion="map"

onRemove="delete"></map>

  </type>

This way, when a contract type changes, the impact can be limited to those operations using this contract type.

## Invoking the Mapper class

## Invoking the ReverseMapper class

# Mapping Reference

Key to defining contracts, mappers and reverse mappers is the definition of the mapping in the ContractsMapping.xml file.

The XML file format is documented by its schema in ContractsMapping.xsd.

## Mapping element (root)

The root element, named “mapping”, has the following attributes and sub elements:

|  |  |
| --- | --- |
| ***mapping*** *element* | |
| @ namespace (required) | The namespace URI for the data contract types. |
| @ defaultBaseClass | Base class to use for contract types when no explicit base class is given. By default System.Object. |
| @ defaultCollectionClass | The class to use for collection types. The collection item type is represented by “{0}”. By default System.Collections.Generic.List<{0}>. |
| <> set [0..\*] | Set elements contain custom information used for code generation. |
| <> reference [1..\*] | References to VS.NET projects or .NET assemblies. |
| <> namespace [0..\*] | Namespaces to include in generated code files. |
| <> type [1..\*] | Mapped type definitions. |

i.e:

<mapping namespace="urn:Ordering.Service.Contract"

         defaultBaseClass="System.Object"

         defaultCollectionClass="System.ComponentModel.BindingList&lt;{0}&gt;"

         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

         xsi:noNamespaceSchemaLocation="ContractsMapping.xsd" >

In this example, all collections will be of type BindingList<T>. All contract types will inherit from System.Object (unless stated differently per type), and have a DataContract namespace of “urn:Ordering.Service.Contract”.

## Reference element

The reference element defines references to VS.NET projects in the current solution, or to compiled .NET assemblies on disk:

|  |  |
| --- | --- |
| ***reference*** *element* | |
| @ project | The name of a project within the current VS.NET solution. |
| @ path | The path to a .NET assembly. |

Only one of both attributes should be given. For instance:

<reference project="Ordering.BusinessLayer"/>

Or

<reference path="..\Ordering.BusinessLayer\bin\debug\Ordering.BusinessLayer.dll"/>

Relative paths are relative towards the current mapping XML file.

Although both examples above should have similar effects, the second one requires the assembly to be compiled and up to date before code generation can be run.

In general, use the path attribute for assemblies of which the code is not included in the current solution.

References must be added using the reference element to all assemblies/projects containing types that are used from within the source domain model. For instance, if the source model contains a type Account with a property of type Money, the assemblies defining the types Account and Money must be referenced in the mapping XML file.

## Namespace element

The namespace element declares the contract namespace, the mapper namespace, and defines additional namespaces to include in generated code.

|  |  |
| --- | --- |
| ***namespace*** *element* | |
| @ name (required) | The namespace name. |
| @ alias | An optional alias for the namespace. |

The contract types are generated in the namespace with alias “Contract”, and therefore, there must be a namespace with this alias. Mapper types are defined in the namespace with alias “Mapper”, or in the default namespace for the mapper project.

Additional namespaces are added as using lines in the generated code.

The following namespace should not be included as they are already included by default:

* System
* System.Collections.Generic
* System.Collections.ObjectModel
* System.Linq
* System.Runtime.Serialization
* System.ComponentModel.DataAnnotations

I.e:

<namespace name="Ordering.Service.Contract" alias="Contract" />

<namespace name="System.ComponentModel" />

<namespace name="System.Globalization" />

## Type element

The type element declares a contract type. A contract type will be generated for each type element in the mapping definition.

|  |  |
| --- | --- |
| ***type*** *element* | |
| @ name (required) | Name of the contract type class (without namespace). |
| @ source (required) | Name of the (business domain) source type class (with namespace). |
| @ abstract | Whether the contract type should be generated abstract. |
| @ baseType | Type this contract type should inherit from. |
| @ typeCondition | Additional condition to differentiate contract subtypes. |
| @ reverseMapping | Supported reverse mapping behavior. One of: none, referenceOnly, update, create, updateAndCreate. By default none. |
| @ dataContractOptions | Additional options to be generated in the [DataContract] attribute of the contract type. |
| <> set [0..\*] | Set elements contain custom information used for code generation. |
| < > attribute [0..\*] | List of attributes to add on the contract type. |
| < > knownSubtype [0..\*] | List of contract subtypes to make this type substitutable by. |
| < > map [0..\*] | Property mapping. |

#### Name & source attributes

The name attribute defines the contract type name to create in the Contract namespace. The source attribute contains the fully qualified name of the source object type (domain type). Several types can be created to map to a single source type. I.e:

  <type name="CustomerListItem" source="MyCorp.CRM.Domain.Customer">

    ...

  </type>

  <type name="CustomerSummary" source="MyCorp.CRM.Domain.Customer">

    ...

  </type>

  <type name="EditableCustomer" source="MyCorp.CRM.Domain.Customer">

    ...

  </type>

#### Other type definition attributes

Additional settings can be defined for the contract type to create:

* The dataContractOptions attribute allows setting additional settings on the generated [DataContract] attribute.
* The abstract attribute (abstract="true") to indicate the generated class should be abstract.
* The baseType attribute to indicate the class this contract class should inherit from.
* The typeCondition attribute allows adding a condition to determine the contract type to map the source type to. This attribute is useful in the case a source type is mapped to an abstract contract type with two or more concrete subtypes. The typeCondition will then allow differentiating between the concrete subtypes to use. The common base type should typically declare the subtypes with the knownSubtype element. I.e:

  <type name="BaseOrder" source="Ordering.Domain.Order" abstract="true">

    <knownSubtype>OpenOrder</knownSubtype>

    <knownSubtype>ClosedOrder</knownSubtype>

    ...

  </type>

  <type name="OpenOrder" source="Ordering.Domain.Order"

        baseType="BaseOrder" typeCondition="source.Status == OrderStatus.Open">

    ...

  </type>

  <type name="ClosedOrder" source="Ordering.Domain.Order"

        baseType="BaseOrder" typeCondition="source.Status != OrderStatus.Open">

    ...

  </type>

#### Mapping-related attributes

The reverseMapping attribute allows for instructing the allowed reverse mapping operations. By default, no reverse mapping code is generated. The options are:

* none: no reverse mapping code is generated (the default).
* referenceOnly: reverse mapping code is generated only to reference the object. For instance, if this option is set on a Customer type, the Customer object will not be updatable or creatable, but orders will be able to reference the correct customer.
* update: the source type can be updated (and referenced) on reverse mapping, but not created. Use this for updatable types that should not support creation or have specific creation operations that does not use this contract type.
* create: the source type can be created (and referenced), but once created can only be referenced, not updated.
* updateAndCreate: the source type can be created, updated and referenced on reverse mapping.

## KnownSubtype element

The KnownSubtype element is a child of the Type element and indicates the *substitutable* subtypes of the containing type.

Consider the following mapping fragment. Two contract types are defined, Product and ComplexProduct that match source types with the same name. ComplexProduct inherits from Product:

  <type name="Product" source="Ordering.Domain.Product">

    <knownSubtype>ComplexProduct</knownSubtype>

    ...

  </type>

  <type name="ComplexProduct" source="Ordering.Domain.ComplexProduct"

        baseType="Product">

    ...

  </type>

Whenever the mapper is to be used to map product into contract types, the following code line can be used:

return new Mapper().MapToProductCollection(result);

Without the marked knownSubtype element in the Product contract type definition, all returned instances will be of type Product, and the only way to obtain a ComplexProduct instance will be to call explicitly the MapToComplexProduct() or MapToComplexProductCollection() mapper methods.

However, If the Product contract type definition contains the marked knownSubtype element, a check is done for each object to be mapped, as whether it could be mapped to the declared subtype. (Note that the typeCondition of the subtype must also allow this.)

With the marked knownSubtype element, the above code line could return a collection containing a mix of Product and ComplexProduct instances.

In addition, the containing contract type will have serialization [KnownType] attribute for use with WCF data contracts ([KnownType(typeof(ComplexProduct))]).

Note that the value of the knownSubtype attribute must be a contract type name declared in the same mapping definition file.

## Map element

The Map element defined the mapping of properties of contract types, and as such, defines the contract type properties themselves.

|  |  |
| --- | --- |
| ***map*** *element* | |
| @ property (required) | The name of the contract type property. |
| @ source | The name or path of the matching property on the source type. By default same as the property value. |
| @ expression | As an alternative to the source, an expression to calculate the value of the property. |
| @ type | The type of the contract property. For collection properties, the type of collection items. By default the same type as the source property. |
| @ conversion | Method to convert source values to contract values. One of: none, map, cast, convert, custom. By default none. |
| @ identifier | Whether this property is (part of) the object identifier. |
| @ updatable | Whether this property can be updated. |
| @ onRemove | For collection properties: action to perform on removed objects. One of: remove, delete, custom. By default remove. |
| @ dataMemberOptions | Additional options to be generated in the [DataMember] attribute of the property. |
| <> set [0..\*] | Set elements contain custom information used for code generation. |
| <> attribute [0..\*] | List of attributes to add on the contract property. |

#### property attribute

The property attribute defines the name of the property, and is also the only mandatory attribute. Hence, the following mapping definition is valid:

  <type name="ProductItem" source="Ordering.DomainLayer.Product">

    <map property="Id"></map>

    <map property="Label"></map>

  </type>

This mapping assumes the Product class in the domain layer had an Id and Label property which are mapped without transformation onto the Id and Label properties of the ProductItem contract type.

#### identifier attribute

Properties that are part of the object identifier (‘primary key’) should have the identifier property set to true. I.e:

    <map property="Id" identifier="true"></map>

#### source attribute

The source attribute can be used when the source of the property is not identical to the property name. The following would for instance create an Id property on the contract type, that maps an ID property on the source type.

    <map property="Id" source="ID"></map>

The source property can also be used to ‘flatten’ the domain structure, and the property path can navigate seamlessly over collections. Consider following examples:

    <map property="CustomerName" source="Customer.Name"></map>

    <map property="ArticleList" source="OrderLines.Article.Name"></map>

The CustomerName is here a classical example of what is called ‘flattening’.

The ArticleList is a more complex situation. Of the order, the source navigates all OrderLines, and for each of them navigates the Article, then the Name of the article. The result is a collection of article names.

Note that there is no special syntax required in the source attribute’s value to navigate collections. Also, if one of the order lines would have its Article property set to null, this line would just be skipped and no ObjectReferenceNotSetException would be thrown.

The source attribute can also contain conditions between square brackets. In this example, the PrincipalRepNames property will contain a collection of names of representatives that have the IsPrincipal flag set to true.

    <map property="PrincipalRepNames"

source="Representatives[r => r.IsPrincipal].Name"></map>

The generator will always assume the result is a collection when the source path navigates through a collection. Even if there would be only one representative of type principal.

#### expression attribute

As an alternative to the source attribute, one can also define an expression attribute. This attribute then contains a C# expression in terms of ‘source’ to retrieve the value of the property. E.g:

    <map property="PrincipalRepName"

expression="source.Representatives  
 .Where(r => r.IsPrincipal)  
 .Select(r => r.Name)  
 .SingleOrDefault()"  
 type="string">  
 </map>  
  
Or:

    <map property="TotalAmount"

expression="source.OrderLines.Sum(l => l.Quantity \* l.UnitPrice)"   
 type="decimal">  
 </map>  
  
When defining an expression, the type attribute must be set as the generator is not able to detect the type of the property when generating code.

#### type attribute

The type attribute defines the type of the property on the contract type. This attribute is only mandatory when the generator cannot determine the property type. For instance, because a conversion method is used, or the property is mapped using an expression.

Note that for collection properties, the type attribute should define the collection member type.

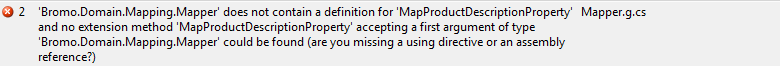
#### conversion attribute

Determines how to convert the value of the source property to the value of the contract type property. By default no conversion is done. The options are:

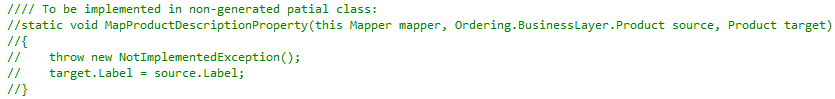
* none: no conversion, the default.
* map: the mapper is used to convert the source property value to the target property value. The type attribute should contain the contract type name to map to.
* cast: casting to the type declared in the type attribute.
* convert: use the System.Convert.ChangeType() method to convert to the type declared in the type attribute.
* custom: you will write your own conversion method. The generator will call a conversion method which you will have to implement. The type attribute is used to determine the property’s type.

Note that whenever a conversion method other than *none* is used, the type attribute must be declared as well.

Whenever the generated code needs to interact with code you have to implement, for instance because custom conversion was chosen, the generated code will simply call a method which is to be implemented. As long as this method is not implemented, the mapper code will have compile exceptions. I.e:



In addition, the generated code will contain the commented out method you need to implement:



You should copy the method declaration to a non-generated source code file and implement it there, where it will not be overwritten by regenerations of the mapper code.

#### updatable attribute

Whether or not the property is updatable. Updatable properties are reversely mapped.

By default properties are considered updatable unless the generation framework is unable to generate the update code.

Setting updatable to false will protect properties from being overwritten.

Setting updatable explicitly to true for properties that are not updatable by default, will trigger the generation of custom reverse mapping code containing of a call to a method you need to implement.

#### onRemove attribute

For properties that represent collections, the onRemove attribute tells what action to take with objects that are removed from the collection. The default is to simply remove the objects. The options are:

* remove: simply removes the value from the collection.
* delete: deletes the object from the object store.
* custom: call a method you will need to implement.

#### dataMemberOptions attribute

The dataMemberOptions attribute allows setting options to the generated [DataMember] attribute on the contract type properties.

## Attribute element

Attribute elements can appear within Type and Map elements. They are used to declare additional code attributes to decorate respectively the generated contract type or one of its properties.

For instance, to add a [Required] attribute on the product’s Label property:

  <type name="Product" source="Ordering.BusinessLayer.Product">

    <map property="Id" identifier="true"></map>

    <map property="Label">

      <attribute>[Required]</attribute>

    </map>

  </type>

## Set element

Set elements can appear within mapping, type, and map elements and allow setting custom properties used by code generation.

|  |  |
| --- | --- |
| ***set*** *element* | |
| @ name (required) | The name of the setting. |
| @ value (required) | Its value. |

Some generation templates use these settings. For instance the *ContractType\_IEntityWithKey.tt* template looks for a “skipIEntityWithKey” setting on the type:

<type name="CustomerItem" source="Ordering.BusinessLayer.Customer">

  <set name="skipIEntityWithKey" value="true"/>

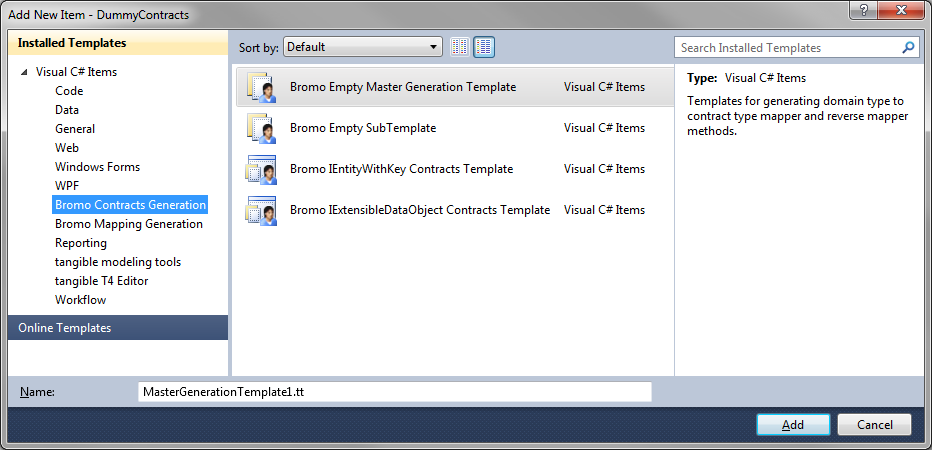
  ...

</type>

# Generation Templates Reference

## Contract Layer Templates

Whenever a new Contracts Library project is created, the project contains a ContractsGenerator.tt template and underneath ContractType.tt, ContractType\_ObjectOverrides.tt and EnumType.tt templates. Additional templates can be added to the Contracts Library project using the *Add New Item* dialog of Visual Studio.NET:



This section describes the purpose of those templates.

### ContractsGenerator.tt

The ContractsGenerator.tt template is a master template and serves only to call the underlying subtemplates. This template creates an instance of a DomainGeneratorSession object to assist in the generation of the subtemplates. See “Understanding master/sub template code generation” under Advanced Topics for more information.

### ContractType.tt

The ContractType.tt template is the template that generates the core of the contract types. The types themselves with their properties.

### EnumType.tt

The EnumType.tt template is the template used to create enumeration contract types.

### ContractType\_ObjectOverrides.tt

This template adds overrides of the Equals, GetHasCode and MemberwiseClone methods, and adds an IsNewInstance method to the contract types. Some of those methods can only be generated for contract types that have identifier properties.

### ContractType\_IEntityWithKey.tt

This template adds IEntityWithKey implementation to contract types, provided the source types have been created by means of an Entity Framework EDMX file. This IEntityWithKey implementation can be used by the reverse mapper’s helper class EntityModelObjectSource which is defined in the Max.Domain.Mapping.Entity.dll assembly.

See also the Getting Started chapter of this documentation.

### ContractType\_IExtensibleDataObject.tt

This template adds IExtensibleDataObject implementation to the contract types.

### ContractModelValidation.tt

Validates the contract model.

## Service Layer / Mapping Templates

A service layer is typically implemented by adding a “Max Mapper Generator Templates Set”, which is a set of 3 templates: MapperGenerator.tt, Mapper.tt and ReverseMapper.tt.

Again, additional templates can be added from VS.NETs “Add New Item” dialog.

### MapperGenerator.tt

This is the master template for the service project. It serves only to generate the subtemplates in its project.

### Mapper.tt

This template generates the mapper code, which is the code used to translate a domain object into a contract object. The generated code consists of extension methods on the Max.Domain.Mapping.Mapper class.

### ReverseMapper.tt

This template generates mapper code for reverse mapping, which is the code to translate a contract object back to a domain object. The generated code consists of extension methods on the Max.Damain.Mapping.ReverseMapper class.

### Mapper\_GetSourcePropertyName.tt

This template adds extension methods to the Max.Domain.Mapping.Mapper class to allow translation of contract type property names into domain type property names for custom use.

### IdentifierBasedObjectSource.tt

### SelectProjections.tt

## Custom templates

### MasterTemplate.tt

### SubTemplate.tt

# Advanced Topics

## Understanding master/sub template code generation

Master templates, typically having a name ending on ~Generator.tt, are regular VS.NET Text Templates that need to have the custom tool property set to “TextTemplatingFileGenerator” (the default), and that contain code similar to the following:

<#@ template debug="false" hostspecific="true" language="C#" #>

<#@ assembly name="Max.Tools.DomainGenerator" #>

<#@ import namespace="System" #>

<#@ import namespace="Max.Tools.DomainGenerator" #>

<#

using (var generator = new DomainGeneratorSession(this, Host))

{

// Setup mapping:

generator.LocalNamespace = "MyContractsProject";

generator.LoadMapping(@".\ContractsMapping.xml");

// Generate all templates within the current project that have the "MaxDomainGeneratorTemplate" CustomTool name:

generator.AutoGenerateLocalTemplates("MaxDomainGeneratorTemplate");

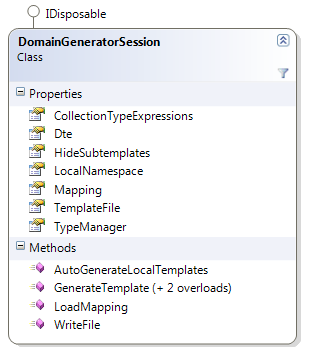
}

#>

<# /\* To generate in the output: \*/ #>

// This file is meant to be empty. //

The template instantiates a DomainGeneratorSession object that assists in generating several subtemplates based on mapping definitions.



Code in the master template ca n call the DomainGeneratorSessions method AutoGenerateLocalTemplates("MaxDomainGeneratorTemplate"). This method searches the current VS.NET project for templates having “MaxDomainGeneratorTemplate” as its “Custom Tool” property value and generates those templates.

When the HideSubtemplates property of the DomainGeneratorSession instance is set (which is by default), then all subtemplates are moved underneath the master template such that they become hidden when collapsing the master template in the Visual Studio.NET Solution Explorer.

As an alternative to the AutoGenerateLocalTemplates() method call, you can also explicitely enumerate the templates to be generated, i.e:

generator.GenerateTemplate("ContractType.tt");

generator.GenerateTemplate("EnumType.tt");

If you want the generated output files to be generated on a different location from the templates, you can explicitely mention the location of the output file. This is useful when referencing templates that are shared over different projects. I.e:

generator.GenerateTemplate(@"\\share\ContractType.tt", "ContractType.g.cs");

generator.GenerateTemplate(@"\\share\EnumType.tt", "EnumType.g.cs");

When called explicitely, subtemplates do not need to be part of the project, and if they are, they do not need to have the custom tool property set with a specific value.

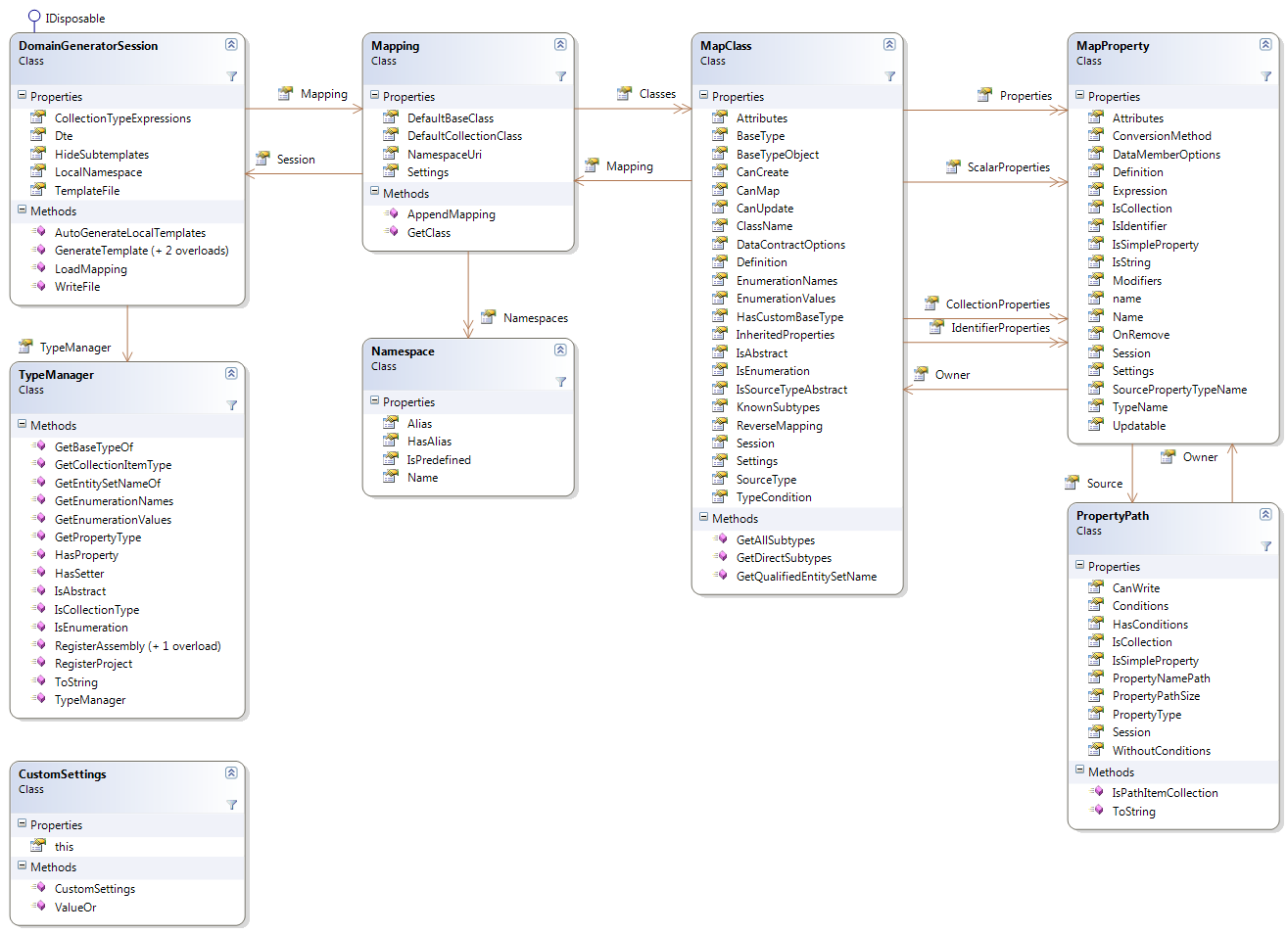
(You may have to add an “<#@ assembly name="EnvDTE, Version=8.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" #>” line on top of the template.)

Subtemplates are also regular VS.NET Text Templates, for two exceptions:

1. subtemplates inherit from a GenerationHostTemplateBase class as you can see in the template directive (the first line) of a subtemplate.
2. when they are part of a project, subtemplates have a custom tool property of “MaxDomainGeneratorTemplate”, or eventueally empty, or something else, but *not* “TextTemplatingFileGenerator” as they are not to be run by the regular VS.NET text templating file generator.

## Understanding generated mapper code

## Extending and Customizing code generation



# Frequently Asked Questions

#### I would like to have a single template for type/mapper/… generation, used by my various projects.

By default, text templates need to be part of the project they generate code for. This means that if you have 10 contracts projects, you will have 10 duplicates of every text template used in the contracts projects, and hence have to maintain and keep in sync (if you want to) those different duplicates.

The text templates generated by Visual Studio must comply this rule. This means that projects own their master templates. However, the sub templates are not necessarily part of the project and can be moved to, for instance, a network share.

In order to generate sub templates that are not part of the project, edit the master template, and replace the call to the AutoGenerateTemplates() method into calls to the GenerateTemplate() overload with as arguments the template file and the output file to generate. Both filenames are absolute, or relative towards the master template:

generator.GenerateTemplate(@"\\share\ContractType.tt", "ContractType.g.cs");

generator.GenerateTemplate(@"\\share\EnumType.tt", "EnumType.g.cs");

#### My domain model uses a collection type that is not detected as a collection type by the generator. How can I make my collection types recognized?

The DomainGeneratorSession object created in the master template has a CollectionTypeExpressions property. This property contains a list of regular expressions of type names to identify as collection types.

By default, the following regular expressions are present:

^(?'itemType'.+)\[\]$"

^.\*Collection\<(?'itemType'.+)\>$

^.\*List\<(?'itemType'.+)\>$

^.\*Set\<(?'itemType'.+)\>$

This means that by default, arrays, and all types ending on Collection, List or Set and having one generic type argument are considered collections.

To have, for instance also types ending on Pool, recognized as collection, add the following line of code in the master template just after creating the DomainGeneratorSession instance:

generator.CollectionTypeExpressions.Add(new Regex(@"^.\*Pool\<(?'itemType'.+)\>$"));

# Known issues

## Declaring conversion method when flattening object graph does not work

In the following example, the cast conversion overrides the pathed source and the result will be code that does NOT support the Customer property to be NULL (while it is exactly MEANT to support that):

<map property="CustomerID" source="Customer.ID" type="int?" conversion="cast"></map>

Solution for now: remove the conversion attribute and make sure no casting or other conversion is required, or choose for ‘custom’ conversion.

## Property source paths have issues with as clause

Although it should have been supported, the as clause in this source expression results in uncompilable mapper code.

<map property="CustomerID" source="Customer.ID as long?"></map>