POCO no Entity Framework 4

Renato Haddad
Microsoft Most Valuable Professional MVP, MCT, MCPD e MCTS
Agosto 2010

Tecnologias
Visual Studio .NET 2010, ASP.NET 4, EF 4

Sumário
Com o lançamento do Visual Studio .NET 2010 ficou fácil gerar os próprios códigos baseados em DSL (Domain Specific Language) e ORM (Modelo de Objeto Relacional), classes e diagramas de acordo com o padrão definido por você.

Introdução
Um dos pontos chave no desenvolvimento de qualquer tipo de aplicação é a produtividade e padronização do código gerado. Isto define o tempo de desenvolvimento, qualidade do código, facilidade de manutenção e entrega do produto final. Isto sem falar nos custos envolvidos, afinal é comum um projeto ser desenvolvido por um time de desenvolvedores, arquitetos, testadores, entre outros.
Uma arquitetura bem definida, seja em camadas ou serviços é o primeiro passo para iniciar a produção de qualquer tipo de projeto de desenvolvimento. O cenário atual não convida à produtividade e padronização de códigos, pois cada desenvolvedor pode e faz o código que bem entender. Já o cenário de fábrica de software tem outras implicações em relação à formação de novos desenvolvedores no time, pois a rotatividade de pessoal é muito grande. Sendo assim, independente do tamanho da equipe é preciso criar um padrão de geração de códigos.

O que o Visual Studio .NET 2010 oferece?
O Visual Studio .NET 2010 tem diversos recursos fantásticos para a geração de códigos. O que vou abordar neste artigo está focado em gerar um modelo de objeto relacional (ORM) a partir de uma base de dados usando o Entity Frameowrk 4. Em seguida, irei padronizar os códigos e gerar as classes de acordo com as entidades existentes no ORM. Em princípio, talvez isto lhe pareça um tanto estranho, mas com o decorrer do artigo você entenderá o meu ponto de vista, as vantagens e desvantagens.

O que é POCO?
POCO significa Plain Old CLR Object, ou seja, é uma maneira de gerar códigos CLR de acordo com um padrão. Este padrão pode ser definido por você em um arquivo do tipo ADO.NET POCO Entity Generator (arquivo.tt). Este tipo de arquivo contém todos os códigos necessários com diversas tags chaves para gerar uma classe a partir de um diagrama ou um modelo. Quando eu me refiro ao diagrama, entenda um ORM, um UML, por exemplo.
A DSL (Domain Specific Language Tools) suporta os templates para transformar os códigos em um arquivo organizado de forma lógica em bloco de acordo o padrão escrito no template. O melhor de tudo é que você consegue debugar o código do template.
O modo mais fácil de usar template é baixar da internet, instalar no VS 2010 automaticamente. O editor mais conhecido é o T4 (Text Template Transformation Toolkit). Então, chegou a hora de instalar os templates e os recursos necessários para desenvolver os exemplos deste artigo. Abra o VS.NET 2010, selecione o menu Tools / Extension Manager para visualizar todas as extensões existentes instaladas. Clique na opção Online Gallery e você visualizará diversas extensões disponíveis na WEB. O mais fácil é digitar T4 na caixa de pesquisa localizada do lado direito superior, isto já filtrada os modelos que precisamos. Veja na figura 1 o filtro aplicado. Não se preocupe se aparecerem outras extensões, afinal é uma pesquisa e constantemente a comunidade inseri templates.

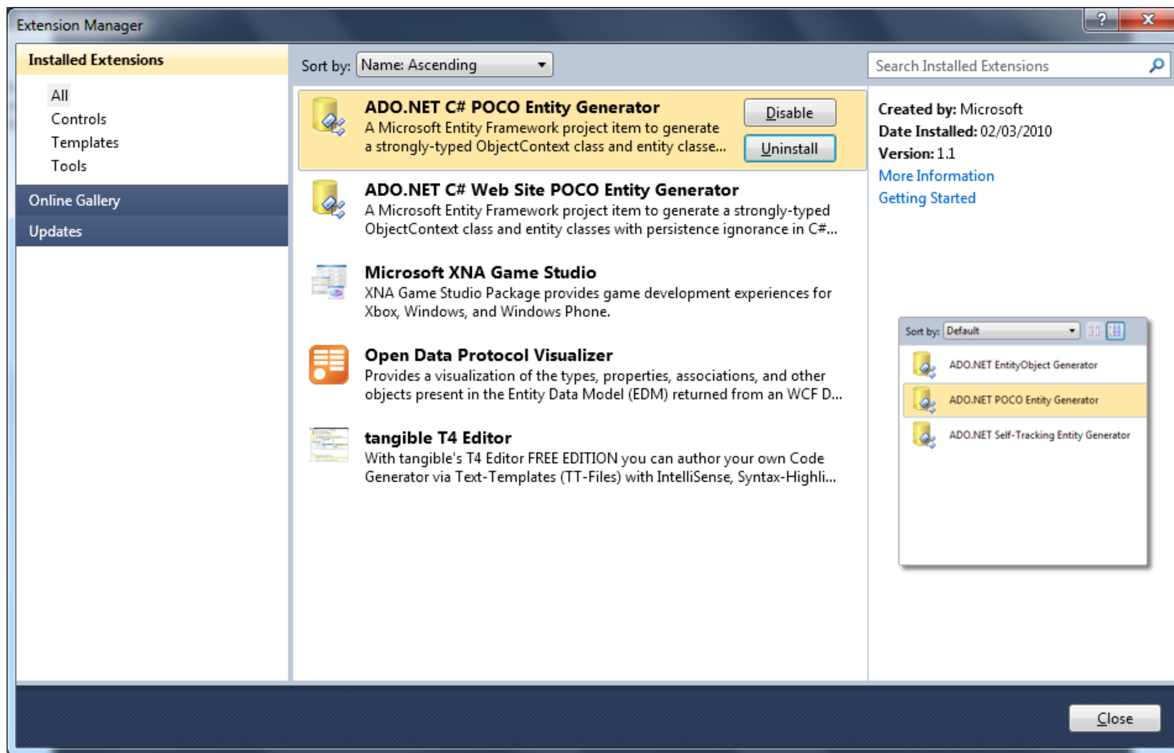

Figura 1 – Templates disponíveis

Instale os templates listados a seguir. Para isto clique no mesmo e selecione o botão download. Automaticamente o VS 2010 faz o download e instala a extensão. Isto é fantástico porque você não tem que instalar a parte, já está tudo integrado.
- tangible T4 Editor
- ADO.NET C# POCO Entity Generator
- ADO.NET C# Web Site POCO Entity Generator

Se você usa VB.NET, baixe mesmo os templates de ADO.NET relacionados ao VB.NET. Pronto, já temos os templates necessários para a aplicação. O melhor de tudo é que você pode ter diversas extensões instaladas e desativar ou desintalar na hora que bem entender. Quando uma extensão é selecionada os botão de Disable/Enable e Uninstall são exibidos e então você pode aplicar a ação. Para finalizar, clique no botão Close.

Projeto

Para criar um projeto com o VS 2010, clique na opção New Project janela Start Page ou no nome File / New / Project. Conforme a figura 2, selecione a linguagem Visual C#, o tipo de projeto ASP.NET Web Application, digite o nome ArtigoEF_POCO, escolha uma pasta e clique no botão OK.

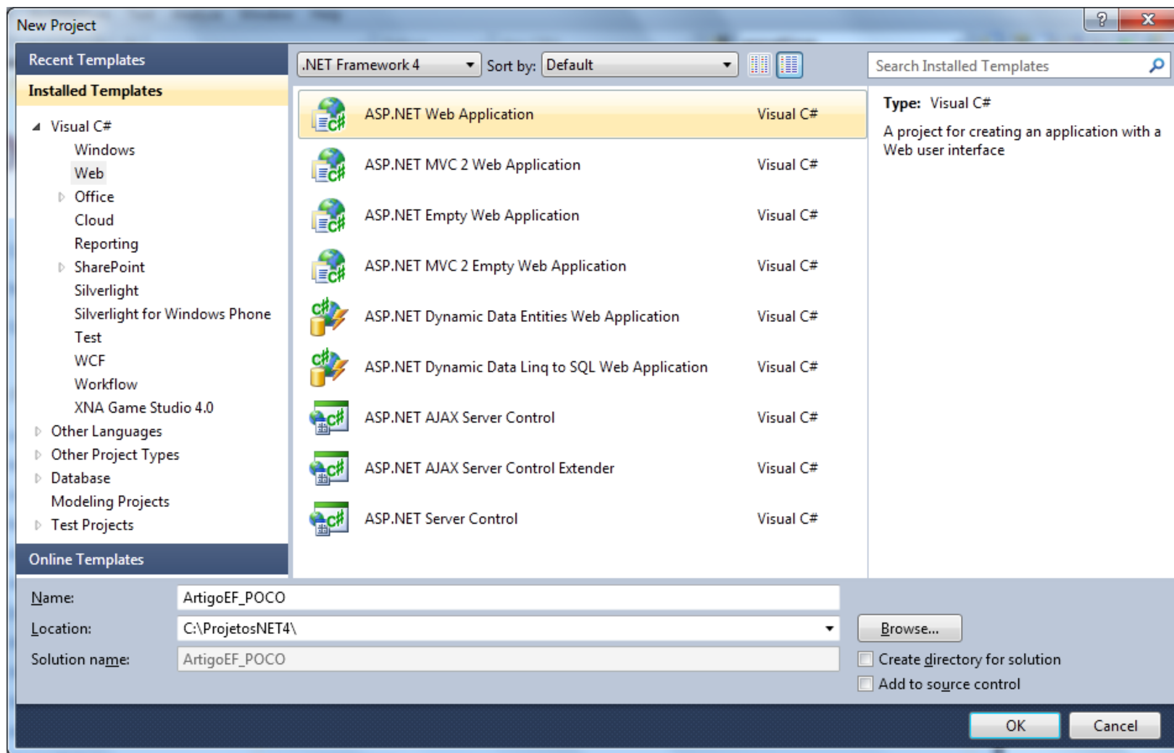

Figura 2 – Novo Projeto ASP.NET

O VS 2010 cria um projeto contendo alguns itens padrão para facilitar a vida do desenvolvedor, por exemplo, formulários aspx, master page e css. Se você quiser saber se a aplicação está rodando, pressione F5 para executar no navegador.
O próximo passo é criar um modelo de objeto relacional (ORM) baseado no Entity Framework 4 e nas tabelas Categories e Products do banco de dados Northwind. Mas, o que é o ORM?
O ORM é um arquivo que contém todas as entidades oriundas de uma fonte de dados, seja ela o SQL Server, o Oracle, ou qualquer outra fonte que já tenha um driver pronto para o Entity Framework. Para o Oracle existe um componente de terceiro em www.devart.com que funciona muito bem. O ORM é dividido em duas partes, sendo a parte lógica e a física. A física é o banco de dados em si que contém as tabelas, consultas e stored procedures. Já a lógica é uma representação dos objetos do banco de dados contidos no ORM. Todo o projeto será baseado no modelo lógico e a enorme vantagem disto é que você poderá desenhar e projetar toda a aplicação para ser usada em qualquer banco de dados.
Para isto, no Solution Explorer, adicione um novo item do tipo ADO.NET Entity Data Model chamado Nwind.edmx, conforme a figura 3. Para você visualizar este template facilmente, aplique um filtro diretamente clicando em Data.

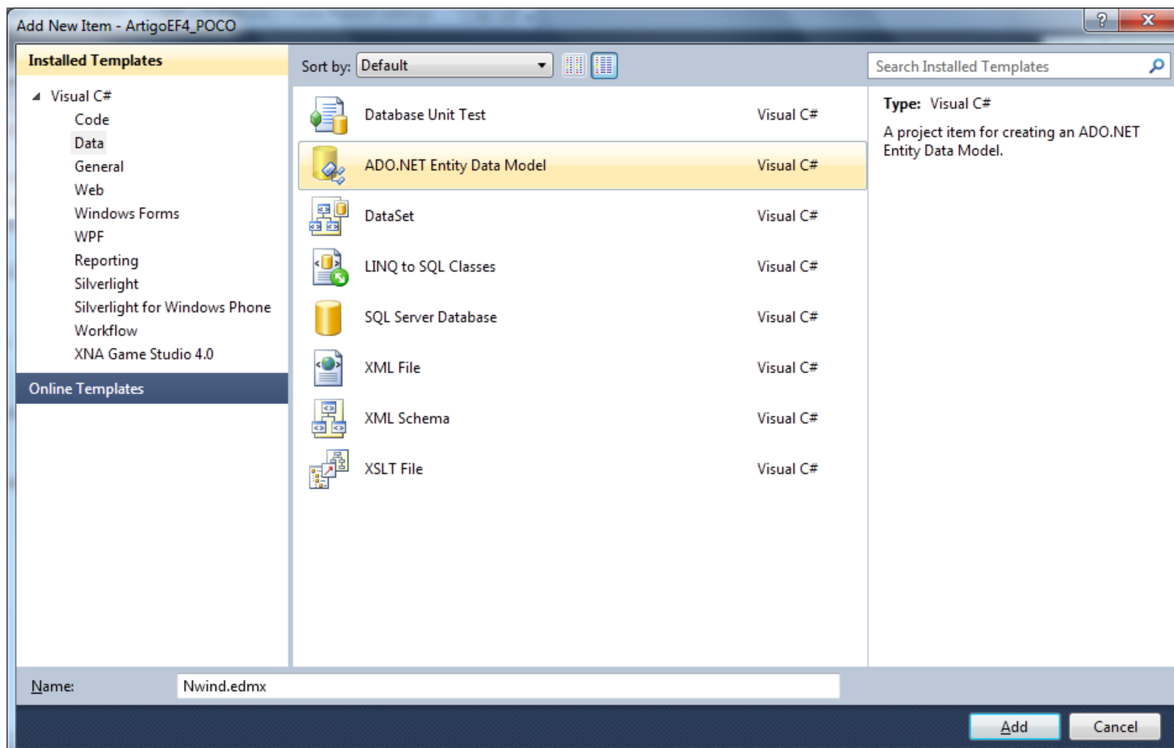

Figura 3 – Arquivo do EDMX

Clique no botão Add e selecione Generated from database, conforme a figura 4. Esta é a única opção, além de ser prática e segura, principalmente se precisar atualizar o modelo caso tenha sido feita alguma alteração no banco de dados, por exemplo, novos objetos, tamanho e tipo de campo.

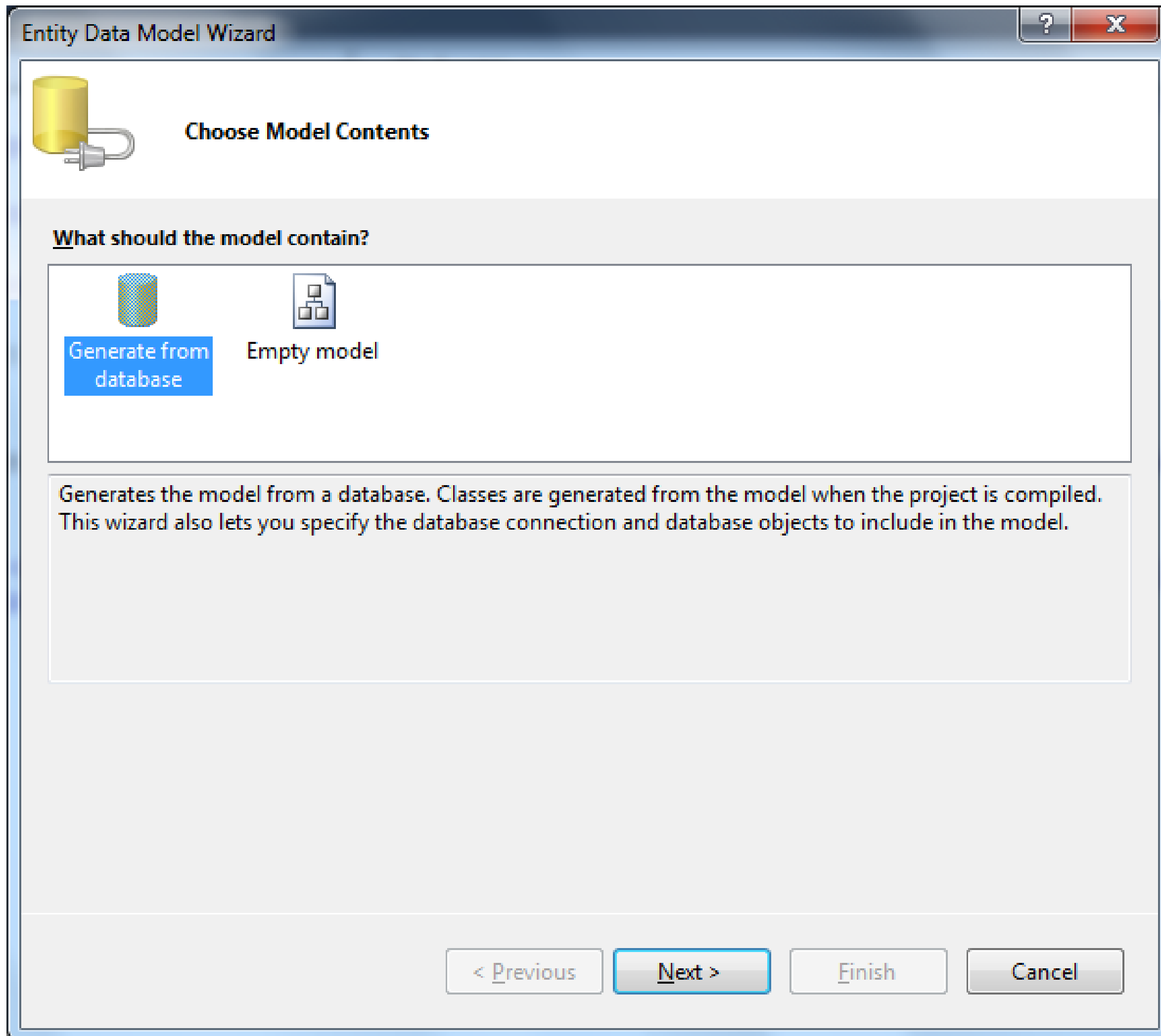

Figura 4 – Passo para gerar o EDMX

Clique no botão Next para o próximo passo. Conforme a figura 5, informe o Data source, ou seja, a fonte de dados, e neste caso selecione o Microsoft SQL Server (SqlClient). Forneça o nome do servidor onde há o SQL Server., assim com as credenciais de login. Por fim, em Select or enter a database name, selecione o Northwind. Pronto, tudo o que você precisa para a conexão está pronto. Clique no botão OK.

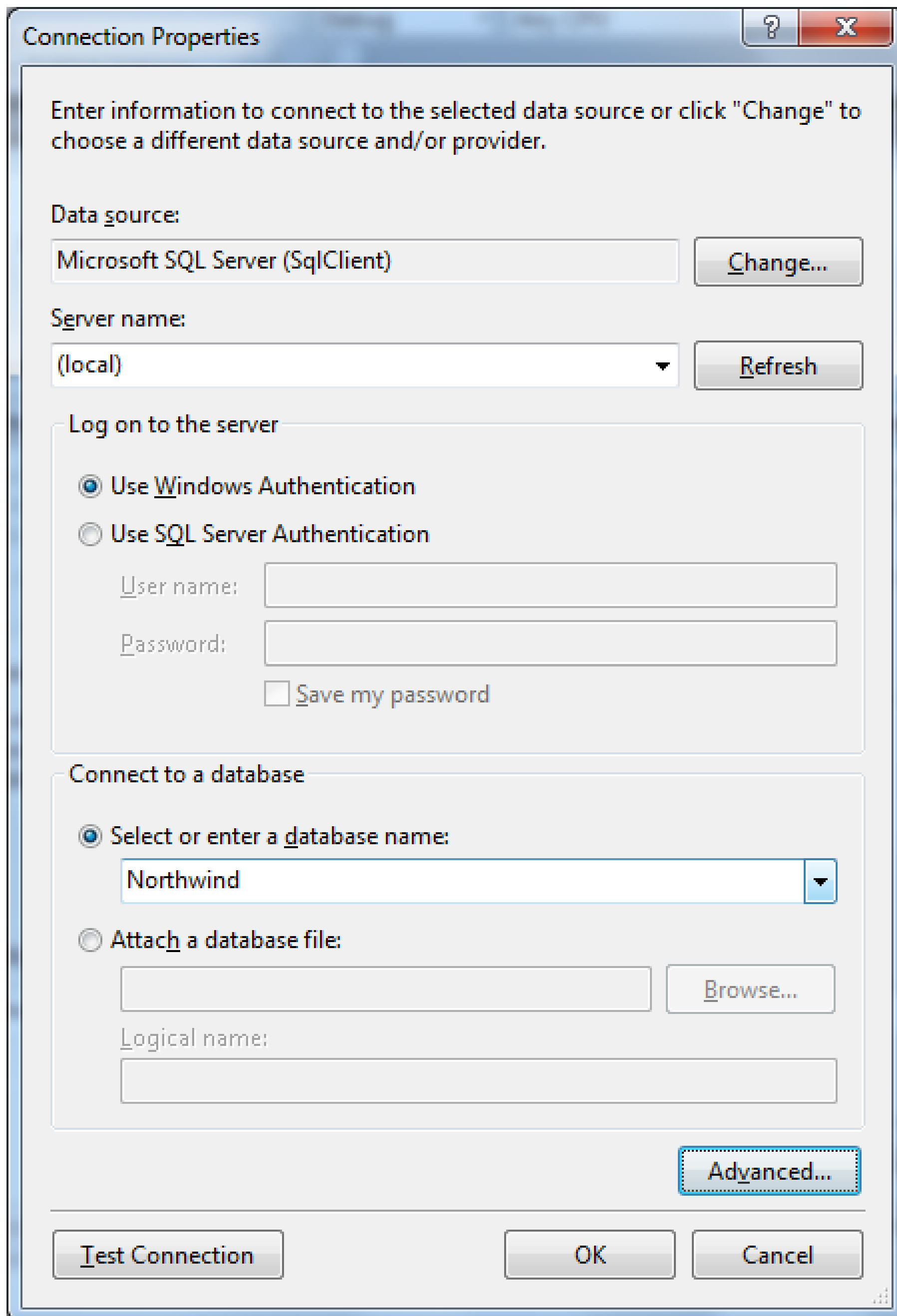

Figura 5 – Dados de acesso ao banco de dados

Em seguida, de acoro com a figura 6, será aberta uma janela indicando a string de conexão que será armazenada no Web.Config, assim como o nome da chave, neste caso é NorthwindEntities. Eu sugiro manter este nome para uma melhor compreensão, mas caso queira alterar fique à vontade. Clique no botão Next.

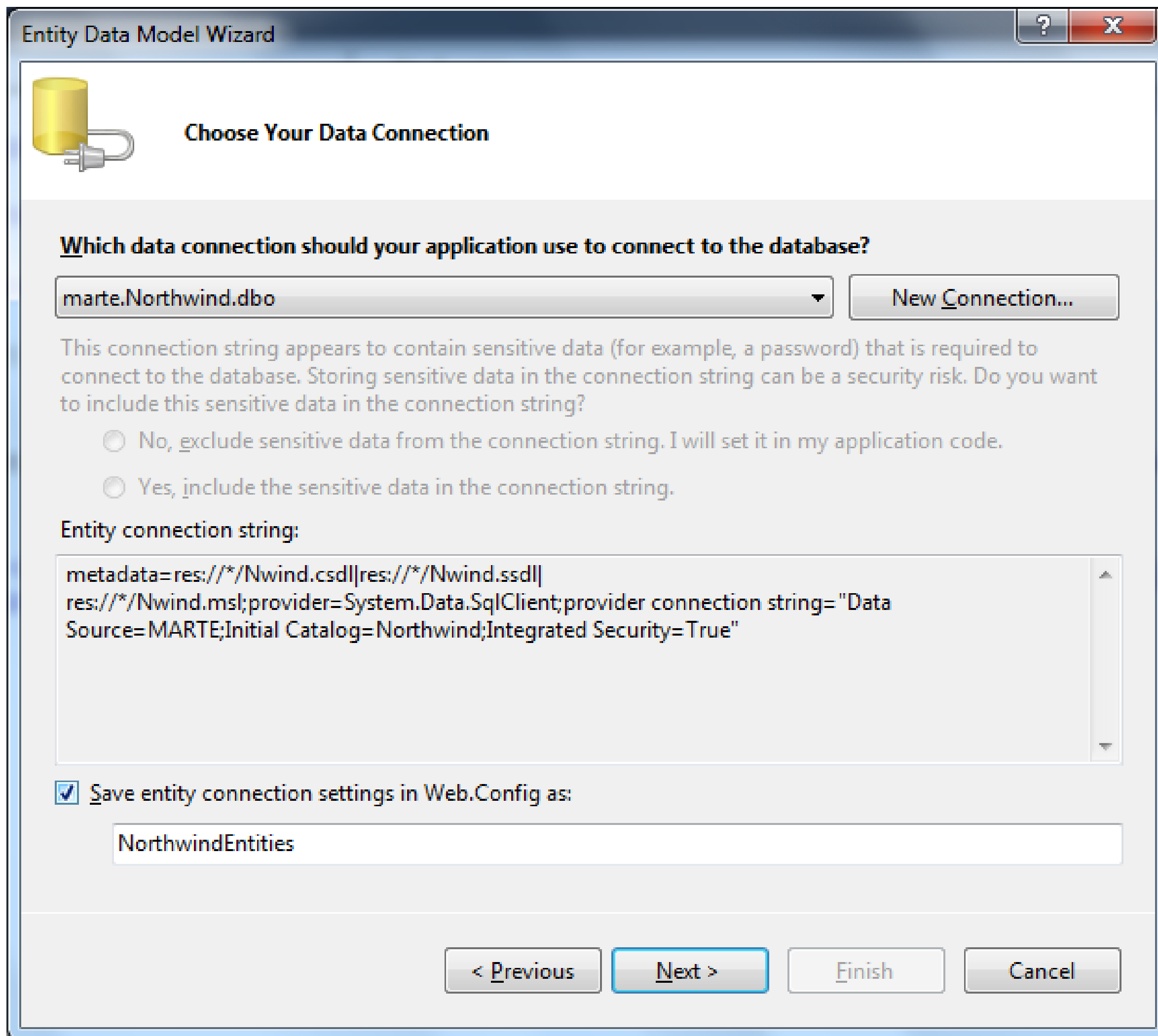

Figura 6 – Dados da conexão no Web.Config

Nesta próxima janela, conforme a figura 7, serão exibidos todos os objetos do banco de dados Northwind selecionado. Em Tables, selecione as tabelas Categories e Products. O nome do Namespace pode deixar o NorthwindModel.

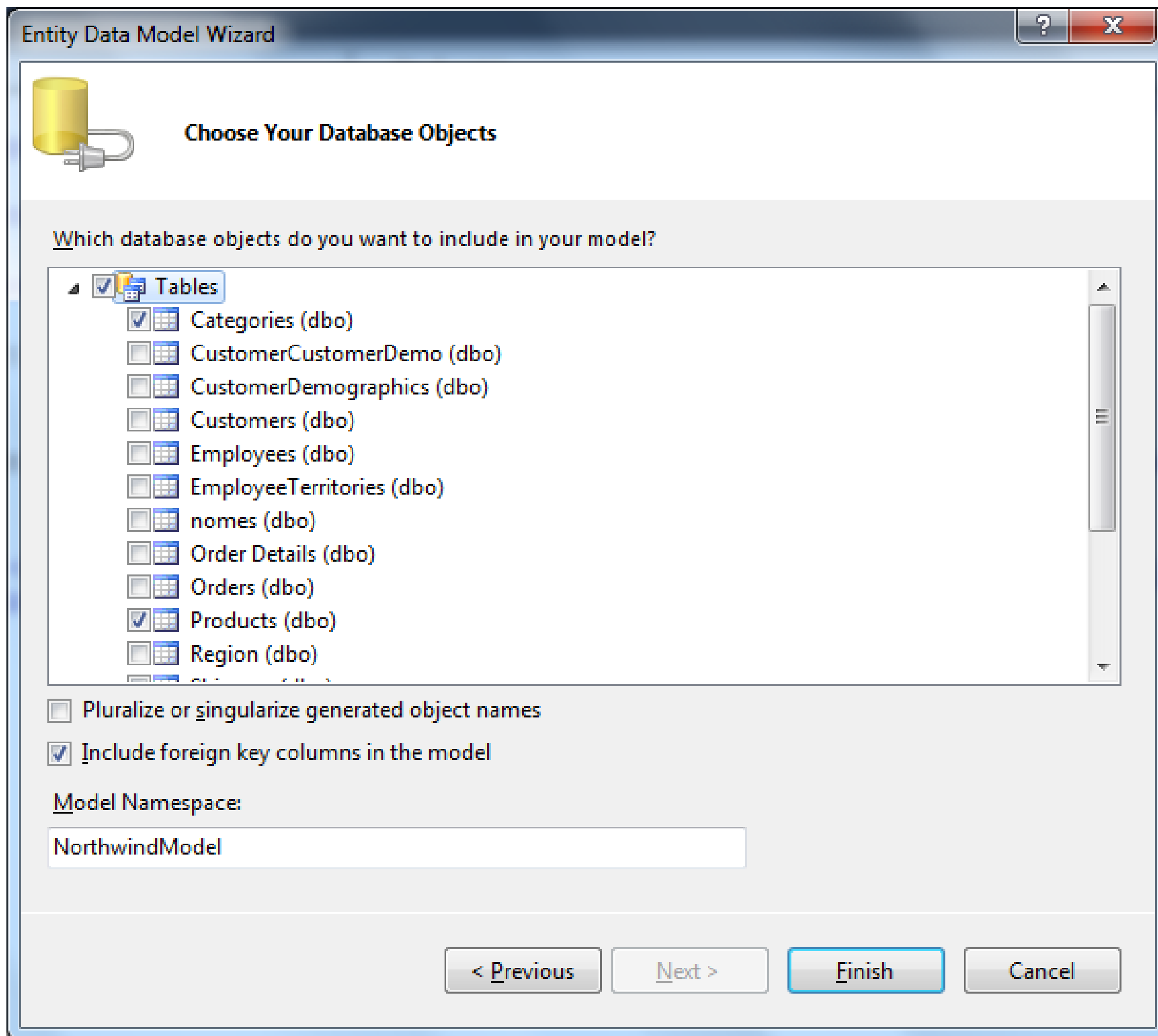

Figura 7 – Objetos existentes no banco de dados

Clique no botão Finish para que o VS 2010 gere o ORM. Veja na figura 8 que o modelo contém exatamente as duas tabelas que foram selecionadas no banco de dados. A partir de agora esqueça os termos tabela e campo, então use classe e propriedades. Pressione CTRL + S para salvar o modelo.

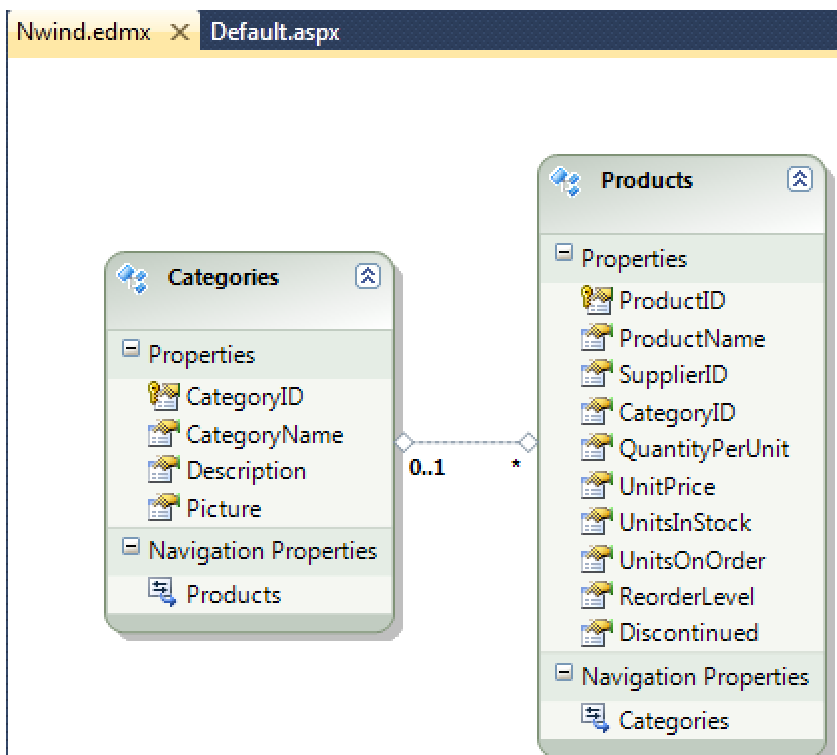

Figura 8 – ORM gerado

O que acontece nos bastidores do VS? Quando este ORM foi criado, automaticamente o VS gerou a classe Nwind.Designer.cs contendo exatamente a definição das classes com as devidas propriedades, e o melhor de tudo é que todo o CRUD (Create, Read, Update e Delete) está pronto. Sugiro que você abra este arquivo e dê uma navegada para perceber o que há no mesmo.

Onde é que entra o T4?

Agora é que vem o foco do artigo. O código gerado pelo ORM está perfeito, funciona muito bem. No entanto, quem criou o código em C# foi o Visual Studio segundo um padrão interno. Em muitos times de desenvolvimento há um padrão a ser seguido, um script a ser usado para facilitar o uso, não que este que foi gerado seja confuso, é apenas uma questão de padronização de código da empresa.
Mas, você deve estar pensando? Abra o código gerado e altere, correto? Errado. Se você fizer isto é o erro mais comum que muitos desenvolvedores tem cometido. Isto porque o código é gerado toda vez que o ORM é alterado, seja uma nova entidade, um novo tipo de campo ou ainda, foi apenas atualizado. A regra é clara: nunca, jamais em hipótese alguma altere o código gerado pelo ORM.
Então, como faço para customizar a classe? A resposta é: crie uma nova classe com o mesmo nome e declare-a como Partial Class e customize o que quiser. No código gerado pelo ORM, todas as classes são do tipo Partial Class, e com isto é possível estender a classe para atender a todas as customizações necessárias.
A questão agora é excluir todo o código gerado pelo ORM automaticamente para qe possamos gerar o nosso código customizado. Para isto, clique no fundo da tela do ORM para selecioná-lo, pressione F4 para exibir a janela de propriedades, e em Code Generation Strategy altere para **None** (conforme a figura 9).

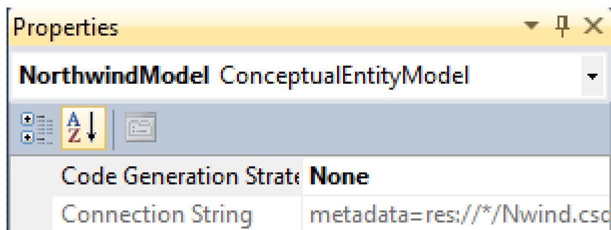

Figura 9 – Janela de propriedades

Em seguida, pressione CTRL + S para salvar o ORM. No Solution Explorer, abra o arquivo Nwind.Designer.cs e veja que o conteúdo está em branco, ou seja, não há nenhuma classe. Cabe ressaltar que se você alterar a propriedade novamente para Default, todo o código automático é gerado novamente. Enfim, deixe-a como None.

Usar o T4 como modelo

Para gerar o código de acordo com o seu padrão, no ORM, clique no fundo da tela com o botão direito e selecione a opção Add Code Generation Item, conforme a figura 10.

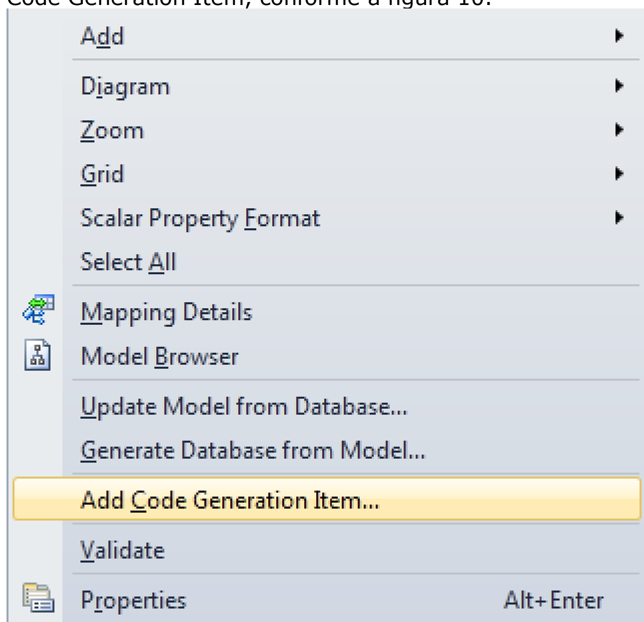

Figura 10 – Gerar código com o T4

Será aberta a janela conforme a figura 11. Clique em Visual C# para exibir todos os templates existentes e instalados na sua máquina. Caso não seja mostrado o template ADO.NET POCO Entity Generator, volte ao início do artigo para instalar via Extension Manager. Este é o template a ser selecionado. O nome do arquivo pode manter o sugerido, Model1.tt. Clique no botão Add para gerar o código.

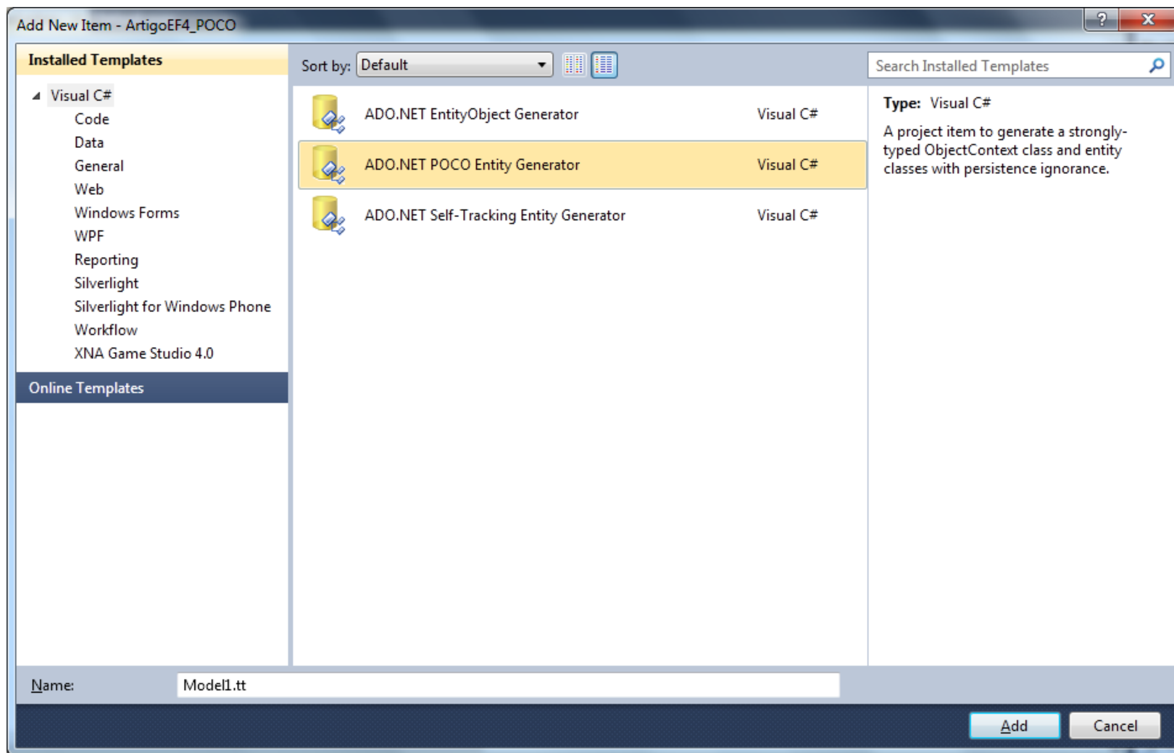

Figura 11 – Usar o ADO.NET POCO

Abra o Solution Explorer e note os seguintes novos arquivos: Model1.Context.tt e Model1.Context.cs. Todo arquivo com extensão .tt é o que contém o modelo em si, e os com extensão .cs é o arquivo gerado a partir do modelo. Tomando como base estes dois arquivo, abra-os e vamos analisar um pouco do que é possível fazer com o template (tt).
Segundo a DSL, num template você pode adicionar diretivas, expressões e instruções. Veja a tabela a seguir com alguns exemplos para entender quais items são:

DSL	Bloco	Descrição
Diretivas	<#@ .. #>	Representa as diretivas a serem processadas pelo engine do template, sendo cinco: template, include, import, assembly e output. <#@ template language="C#" debug="false" hostspecific="true"#> <#@ include file="EF.Utility.CS.ttinclude"#> <#@ output extension=".cs"#>

Instruções	<# textos #>	São textos gerados exatamente com estão escritos. Isto é usado, por exemplo, na declaração da classe (public partial class)
		partial class <#=code.Escape(container)#>
		ou
		<#
		if (!String.IsNullOrEmpty(namespaceName))
		{
		#>

Expressões	<#= expressão #>	São expressões usadas para palavras chave ou nome de variável. Por exemplo <#=code.Escape(container)#> ou <#=container.Name#>

Inicialmente no arquivo **Model1.Context.tt**.há diversas linhas comentadas. Em seguida, você irá notar uma tag especial para abrir e fechar o bloco de diretivas para informar ao engine do template quais são os dados que queremos usar, por exemplo, a linguagem C#, o debug e a extensão do arquivo de saída.

```
<#@ template language="C#" debug="false" hostspecific="true"#>
<#@ include file="EF.Utility.CS.ttinclude"#>
<#@ output extension=".cs"#>
```

Em seguida temos um conjunto de instruções para o engine do template contendo os códigos para a geração e os metadados. Note que existem duas variáveis criadas (inputFile e namespaceName) que serão utilizadas em outras partes do código do template.

```
<#
CodeGenerationTools code = new CodeGenerationTools(this);
MetadataTools ef = new MetadataTools(this);
MetadataLoader loader = new MetadataLoader(this);
CodeRegion region = new CodeRegion(this);

string inputFile = @"Nwind.edmx";
```

```
EdmItemCollection ItemCollection = loader.CreateEdmItemCollection(inputFile);
string namespaceName = code.VsNamespaceSuggestion();

EntityContainer container = ItemCollection.GetItems<EntityContainer>().FirstOrDefault();
if (container == null)
{
    return "// No EntityContainer exists in the model, so no code was generated";
}
#>
```

Em seguida temos os comentários e as declarações dos usings que serão inseridos no arquivo .cs a ser gerado. É exatamente neste ponto (aproximadamente na linha 41) que declaramos todas as referências do nosso código, ou seja, se você tem a aplicação que utiliza outas referências, por exemplo camada de negócios, camada de dados, helpers, etc, é aqui que deverão ser declarados. Enfim, creio que com estas explicações abordadas anteriormente no artigo você já consiga identificar os demais códigos.
Como este é o arquivo Model1.Context.tt, qdo você alterar qualquer coisa no mesmo, salve-o (CTRL + S) para que seja imediatamente gerado o arquivo Model1.Context.cs, o qual contém as declarações dos objetos, toda a parte de string de conexão com o banco de dados e as declarações dos objetos Categories e Products, neste caso. O interessante é que qualquer objeto que terá um CRUD usa a classe ObjectSet, o qual recebe a entidade como parâmetro e monta o CRUD em tempo de execução.

```
<#=Accessibility.ForReadOnlyProperty(entitySet)#> ObjectSet<<#=code.Escape(entitySet.ElementType)#>>
<#=code.Escape(entitySet)#>
{
    get { return <#=code.FieldName(entitySet) #>  ?? (<#=code.FieldName(entitySet)#> =
CreateObjectSet<<#=code.Escape(entitySet.ElementType)#>>("<#=entitySet.Name#>")); }
}
private ObjectSet<<#=code.Escape(entitySet.ElementType)#>> <#=code.FieldName(entitySet)#>;

Este código do modelo acima irá gerar o seguinte código no .cs:
public ObjectSet<Categories> Categories
{
    get { return _categories  ?? (_categories = CreateObjectSet<Categories>("Categories")); }
}
```

Note no solution Explorer que foi gerado um outro modelo chamado Model1.tt. Para cada entidade existente no ORM foi gerada uma classe, neste caso Categories.cs e Products.cs. Basicamente o modelo do Model1.tt irá gerar este arquivo .cs. A classe FixupCollection é responsável por limpar e inserir items dinamicamente. No entanto, como esta classe herda a ObservableCollection (novidade no .Net 4.0), o qual representa uma coleção de dados dinâmicos que fornece notificações quando houver uma inclusão, exclusão ou atualização da lista. No método ClearItems é usado o Generic List<t> que usa o Action .ForEach para aplicar a ação de remover items da lista. O Action é fantástico porque ele dispara um delegate passando um objeto como parâmetro. O Action não é nenhuma novidade pois existe no C# 3 e no VB9, e isto reduz drasticamente a quantidade de códigos. Já o método InsertItem usa o Contains do LINQ para verificar se o item existe na coleção, e caso não exista, o item é adicionado à coleção.

```
public class FixupCollection<T> : ObservableCollection<T>
{
    protected override void ClearItems()
    {
        new List<T>(this).ForEach(t => Remove(t));
    }

    protected override void InsertItem(int index, T item)
    {
        if (!this.Contains(item))
        {
            base.InsertItem(index, item);
        }
    }
}
```

O arquivo Categories.cs gerado baseado no Model1.tt basicamente contém a declaração de todas as propriedades da classe, a interface ICollection para que sejam relacionados todos os produtos à categoria atual, afinal, uma categoria pode conter diversos produtos, e neste caso, o modelo já identifica isto automaticamente. É exatamente neste momento que a classe FixupCollection existente no Model1.cs é usada. Enfim, o que quero deixar claro é que se você tiver um ORM bem definido com todos os relacionamentos entre as entidades, o modelo se encarrega de montar todo o código.
O mesmo ocorre com a classe Product.cs gerada, o qual contém todas as propriedades da entidade e a navegação para identificar a qual categoria o respectivo produto pertence.
O arquivo Model1.tt é extenso, com cerca de 800 linhas. Então, vamos alterar um pedaço do código para visualizar o que o modelo gerou. Vá para a linha 723 onde iniciam-se os comentários e adicione a seguinte linha em português:

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated from a template.
//     Este código foi gerado para o artigo de POCO !!!
```

```
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------
```

Em seguida, na lista de using adicione o using para usar o StringBuilder:

```
using System.Text;
```

Se houver outros usings a serem declarados, este looping se encarrega disto. Você não precisa digitar isto, pois já existe no modelo.

```
<#=String.Join(String.Empty, extraUsings.Select(u => "using " + u + ";" + Environment.NewLine).ToArray())#>
<#+
    fileManager.EndBlock();
}
```

Pressione CTRL + S para salvar o modelo e abra os arquivos Categories.cs e Products.cs para visualizar que o comentário e o using que inserimos foram adicionados.
Como você pode ver até aqui, o modelo é uma estrutura lógica que contém blocos de tags especiais, linguagem de programação para montar as classes dinamicamente. A partir disto você pode montar um modelo exatamente de acordo com o padrão de codificação da sua empresa.
A DSL nos ajuda em montar estes padrões com as tags existentes na DSL. Só para saber, o Visual Studio .NET 2010 permite criar 7 diferentes tipos de diagramas (UML, Layer e Directed Graph) graficamente, conforme a figura 12.

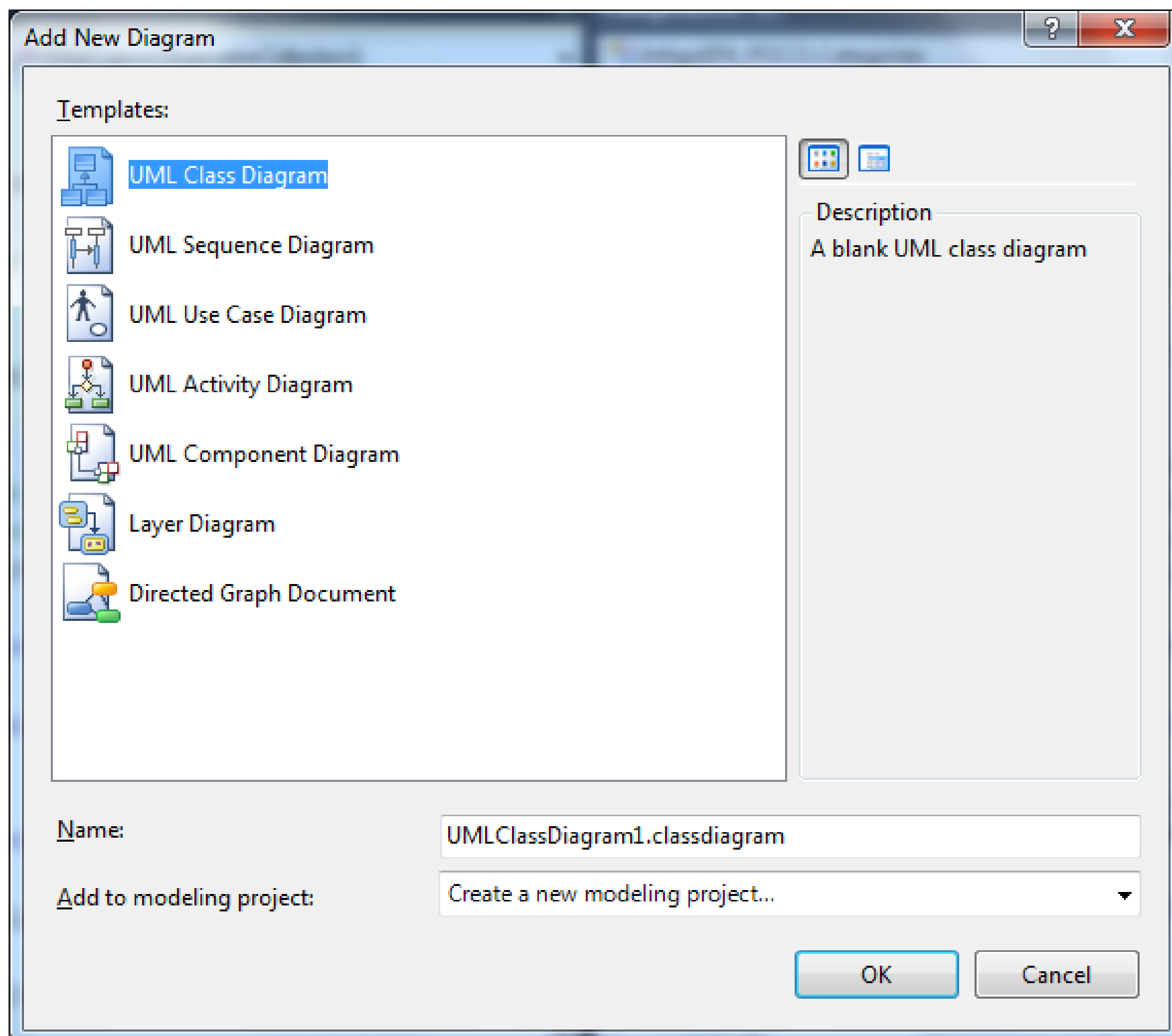

Figura 12 – Diagramas do VS 2010

Neste caso dos diagramas é possível desenvolver um template para que gere um código baseado no gráfico montado.

Para Saber Mais

- Walkthrough: Creating and Running Text Templates http://msdn.microsoft.com/en-us/library/bb126484.aspx
Creating Domain-Specific Languages http://msdn.microsoft.com/en-us/library/bb126259(v=VS.80).aspx
Requirements for Creating POCO Proxies(Entity Framework) - http://msdn.microsoft.com/en-us/library/dd468057.aspx

Working with POCO Entities (Entity Framework) http://msdn.microsoft.com/en-us/library/dd456853.aspx
Tracking Changes in POCO Entities(Entity Framework) http://msdn.microsoft.com/en-us/library/dd456848.aspx

Roadmap

A tecnologia usada neste artigo foi DSL para a geração de códigos com o modelo T4. Você precisa usar o Visual Studio 2010, fazer download dos templates de POCO citados do início do artigo.

Considerações Finais

Um dos focos do VS 2010 é promover a produtividade e oferecer aos arquitetos e desenvolvedores meios de gerar códigos padronizados. Aconselho que você estude o Entity Framework 4, os modelos de POCO, DSL e a aplicabilidade nos projetos.

Sobre o Autor

Renato Haddad (rehaddad@msn.com – www.renatohaddad.com) é MVP, MCT, MCPD e MCTS, palestrante em eventos da Microsoft em diversos países, ministra treinamentos focados em produtividade com o VS.NET 2010, ASP.NET 4, Entity Framework, Reporting Services e Windows Mobile. Visite o blog http://weblogs.asp.net/renatohaddad e a loja de treinamentos www.renatohaddad.com/loja