```
<#
//     You can use this text template to customize object layer code generation for
//     applications that use the Entity Framework. The template generates code based on an .edmx file.
//     Before using this template, note the following:
//
//         *The name of the text template file will determine the name of the code file it generates.
//           For example, if the text template is named TextTemplate.tt, the generated file will be named
//           TextTemplate.vb or TextTemplate.cs.
//         *The Custom Tool property of the targeted .edmx file must be empty. For more information,
//          see .edmx File Properties (http://go.microsoft.com/fwlink/?LinkId=139299).
//         *The SourceCsdlPath initialization below must be set to one of the following:
//                 1) the path of the targeted .edmx or .csdl file
//                 2) the path of the targeted .edmx or .csdl file relative to the template path
//
//     For more detailed information about using this template, see
//     How to: Customize Object Layer Code Generation (http://go.microsoft.com/fwlink/?LinkId=139297).
//     For general information about text templates, see
//     Generating Artifacts by Using Text Templates (http://go.microsoft.com/fwlink/?LinkId=139298)
#>
<#@ template language="C#" debug="false" hostspecific="true"#>
<#@ include file="EF.Utility.CS.ttinclude"#><#@
 output extension=".cs"#><#

DefineMetadata();

UserSettings userSettings =
            new UserSettings
            {
                        SourceCsdlPath = @"SchoolEntities.edmx",
                        ReferenceCsdlPaths = new string[] {},
                        FullyQualifySystemTypes = true,
                        CreateContextAddToMethods = true,
                        CamelCaseFields = false,
            };

ApplyUserSettings(userSettings);
if(Errors.HasErrors)
{
            return String.Empty;
}
```

```
MetadataLoader loader = new MetadataLoader(this);
MetadataTools ef = new MetadataTools(this);
CodeRegion region = new CodeRegion(this);
CodeGenerationTools code = new CodeGenerationTools(this){FullyQualifySystemTypes = userSettings.FullyQualifySystemTypes, CamelCaseFields = userSettings.CamelCaseFields};

ItemCollection = loader.CreateEdmItemCollection(SourceCsdlPath, ReferenceCsdlPaths.ToArray());
ModelNamespace = loader.GetModelNamespace(SourceCsdlPath);
string namespaceName = GetNamespaceName(code);
UpdateObjectNamespaceMap(namespaceName);

#>
//------------------------------------------------------------------------------
// <auto-generated>
// <#=CodeGenerationTools.GetResourceString("Template_GeneratedCodeCommentLine1")#>
//
// <#=CodeGenerationTools.GetResourceString("Template_GeneratedCodeCommentLine2")#>
// <#=CodeGenerationTools.GetResourceString("Template_GeneratedCodeCommentLine3")#>
// </auto-generated>
//------------------------------------------------------------------------------

using System;
using System.ComponentModel;
using System.Data.EntityClient;
using System.Data.Objects;
using System.Data.Objects.DataClasses;
using System.Linq;
using System.Runtime.Serialization;
using System.Xml.Serialization;

[assembly: EdmSchemaAttribute()]
<#
        ////////
        ////////  Write Relationship Attributes
        ////////
        region.Begin(CodeGenerationTools.GetResourceString("Template_RegionRelationships"));
        bool first = true;
        foreach (AssociationType association in GetSourceSchemaTypes<AssociationType>())
        {
            if (first)
            {
                WriteLine(string.Empty);
```

```
                                first = false;
                        }
#>
[assembly: EdmRelationshipAttribute("<#=association.NamespaceName#>", "<#=association.Name#>", "<#=EndName(association, 0)#>", <#=EndMultiplicity(association, 0, code)#>,
typeof(<#=EscapeEndTypeName(association, 0, code)#>), "<#=EndName(association, 1)#>", <#=EndMultiplicity(association, 1, code)#>,
typeof(<#=EscapeEndTypeName(association, 1, code)#>)<#=code.StringBefore(", ", association.IsForeignKey ? "true" : null)#>)]
<#
            }
            region.End();

            if (!String.IsNullOrEmpty(namespaceName))
            {
#>
namespace <#=namespaceName#>
{
<#
                PushIndent(CodeRegion.GetIndent(1));
            }

            ////////
            ////////  Write EntityContainer and ObjectContext classes.
            ////////

            region.Begin(CodeGenerationTools.GetResourceString("Template_RegionContexts"));
            foreach (EntityContainer container in GetSourceSchemaTypes<EntityContainer>())
            {
#>

/// <summary>
/// <#=SummaryComment(container)#>
/// </summary><#=LongDescriptionCommentElement(container, region.CurrentIndentLevel)#>
<#=Accessibility.ForType(container)#> partial class <#=code.Escape(container)#> : ObjectContext
{
            #region <#=CodeGenerationTools.GetResourceString("Template_RegionConstructors")#>

            /// <summary>
            /// <#=String.Format(CultureInfo.CurrentCulture, CodeGenerationTools.GetResourceString("Template_ContextDefaultCtorComment"), container.Name, container.Name)#>
            /// </summary>
            public <#=code.Escape(container)#>() : base("name=<#=container.Name#>", "<#=container.Name#>")
            {
<#
```

```
                WriteLazyLoadingEnabled(container);
#>
                OnContextCreated();
        }

        /// <summary>
        /// <#=String.Format(CultureInfo.CurrentCulture, CodeGenerationTools.GetResourceString("Template_ContextCommonCtorComment"), container.Name)#>
        /// </summary>
        public <#=code.Escape(container)#>(string connectionString) : base(connectionString, "<#=container.Name#>")
        {
<#
                WriteLazyLoadingEnabled(container);
#>
                OnContextCreated();
        }

        /// <summary>
        /// <#=String.Format(CultureInfo.CurrentCulture, CodeGenerationTools.GetResourceString("Template_ContextCommonCtorComment"), container.Name)#>
        /// </summary>
        public <#=code.Escape(container)#>(EntityConnection connection) : base(connection, "<#=container.Name#>")
        {
<#
                WriteLazyLoadingEnabled(container);
#>
                OnContextCreated();
        }

        #endregion

        #region <#=CodeGenerationTools.GetResourceString("Template_RegionPartialMethods")#>

        partial void OnContextCreated();

        #endregion

<#
                ////////
                ////////  Write EntityContainer and ObjectContext ObjectSet properties.
                ////////
                region.Begin(CodeGenerationTools.GetResourceString("Template_RegionObjectSetProperties"));
                foreach (EntitySet set in container.BaseEntitySets.OfType<EntitySet>())
```

```
                {
                        VerifyEntityTypeAndSetAccessibilityCompatability(set);
#>

        /// <summary>
        /// <#=SummaryComment(set)#>
        /// </summary><#=LongDescriptionCommentElement(set, region.CurrentIndentLevel)#>
        <#=code.SpaceAfter(NewModifier(set))#><#=Accessibility.ForReadOnlyProperty(set)#> ObjectSet<<#=MultiSchemaEscape(set.ElementType, code)#>>
<#=code.Escape(set)#>
        {
                get
                {
                        if ((<#=code.FieldName(set)#> == null))
                        {
                                <#=code.FieldName(set)#> = base.CreateObjectSet<<#=MultiSchemaEscape(set.ElementType, code)#>>("<#=set.Name#>");
                        }
                        return <#=code.FieldName(set)#>;
                }
        }
        private ObjectSet<<#=MultiSchemaEscape(set.ElementType, code)#>> <#=code.FieldName(set)#>;
<#
                }
                region.End();

                ////////
                ////////  Write EntityContainer and ObjectContext AddTo&lt;EntitySet&gt; methods.
                ////////
                ////////  AddTo methods are no longer necessary since the EntitySet properties return
                ////////  an ObjectSet&lt;T&gt; object,  which has already has an Add method.
                ////////
                ////////  AddTo methods are generated here for backwards compatibility reasons only.
                ////////  Set the CreateContextAddToMethods property of the UserSettings object to false
                ////////  to turn off generation of the AddTo methods.
                ////////
                region.Begin(CodeGenerationTools.GetResourceString("Template_RegionAddToMethods"));
                IEnumerable<EntitySet> addToMethods = CreateContextAddToMethods.Value ? container.BaseEntitySets.OfType<EntitySet>() : Enumerable.Empty<EntitySet>();
                foreach (EntitySet set in addToMethods)
                {
                        string parameterName = code.Escape(FixParameterName(set.ElementType.Name, code));
#>
```

```
        /// <summary>
        /// <#=String.Format(CultureInfo.CurrentCulture, CodeGenerationTools.GetResourceString("Template_GenCommentAddToMethodCs"), set.Name)#>
        /// </summary>
        <#=Accessibility.ForType(set.ElementType)#> void AddTo<#=set.Name#>(<#=MultiSchemaEscape(set.ElementType, code)#> <#=parameterName#>)
        {
                base.AddObject("<#=set.Name#>", <#=parameterName#>);
        }
<#
                }
                region.End();

                ////////
                ////////  Write EntityContainer and ObjectContext Function Import methods.
                ////////
                region.Begin(CodeGenerationTools.GetResourceString("Template_RegionFunctionImports"));
                foreach (EdmFunction edmFunction in container.FunctionImports)
                {

                        IEnumerable<FunctionImportParameter> parameters = FunctionImportParameter.Create(edmFunction.Parameters, code, ef);
                        string paramList = string.Join(", ", parameters.Select(p => p.FunctionParameterType + " " + p.FunctionParameterName).ToArray());
                        TypeUsage returnType = edmFunction.ReturnParameters.Count == 0 ? null : ef.GetElementType(edmFunction.ReturnParameters[0].TypeUsage);
                        if (edmFunction.IsComposableAttribute)
                        {
#>

        /// <summary>
        /// <#=SummaryComment(edmFunction)#>
        /// </summary><#=LongDescriptionCommentElement(edmFunction, region.CurrentIndentLevel)#><#=ParameterComments(parameters.Select(p => new Tuple<string,
string>(p.RawFunctionParameterName, SummaryComment(p.Source))), region.CurrentIndentLevel)#>
        [EdmFunction("<#=edmFunction.NamespaceName#>", "<#=edmFunction.Name#>")]
        <#=code.SpaceAfter(NewModifier(edmFunction))#><#=Accessibility.ForMethod(edmFunction)#> <#="IQueryable<" + MultiSchemaEscape(returnType, code) + ">"#>
<#=code.Escape(edmFunction)#>(<#=paramList#>)
        {
<#
                                WriteFunctionParameters(parameters);
#>
                return base.CreateQuery<<#=MultiSchemaEscape(returnType, code)#>>("[<#=edmFunction.NamespaceName#>].[<#=edmFunction.Name#>](<#=string.Join(", ",
parameters.Select(p => "@" + p.EsqlParameterName).ToArray())#>)"<#=code.StringBefore(", ", string.Join(", ", parameters.Select(p => p.ExecuteParameterName).ToArray()))#>);
        }
<#
                        }
```

```
                                    else
                                    {
#>

        /// <summary>
        /// <#=SummaryComment(edmFunction)#>
        /// </summary><#=LongDescriptionCommentElement(edmFunction, region.CurrentIndentLevel)#><#=ParameterComments(parameters.Select(p => new Tuple<string,
string>(p.RawFunctionParameterName, SummaryComment(p.Source))), region.CurrentIndentLevel)#>
        <#=code.SpaceAfter(NewModifier(edmFunction))#><#=Accessibility.ForMethod(edmFunction)#> <#=returnType == null ? "int" : "ObjectResult<" +
MultiSchemaEscape(returnType, code) + ">"#> <#=code.Escape(edmFunction)#>(<#=paramList#>)
        {
<#
                                        WriteFunctionParameters(parameters);
#>
            return base.ExecuteFunction<#=returnType == null ? "" : "<" + MultiSchemaEscape(returnType, code) + ">"#>("<#=edmFunction.Name#>"<#=code.StringBefore(",
", string.Join(", ", parameters.Select(p => p.ExecuteParameterName).ToArray()))#>);
        }
<#
                                        if(returnType != null && returnType.EdmType.BuiltInTypeKind == BuiltInTypeKind.EntityType)
                                        {
#>
        /// <summary>
        /// <#=SummaryComment(edmFunction)#>
        /// </summary><#=LongDescriptionCommentElement(edmFunction, region.CurrentIndentLevel)#>
        /// <param name="mergeOption"></param><#=ParameterComments(parameters.Select(p => new Tuple<string, string>(p.RawFunctionParameterName,
SummaryComment(p.Source))), region.CurrentIndentLevel)#>
        <#=code.SpaceAfter(NewModifier(edmFunction))#><#=Accessibility.ForMethod(edmFunction)#> <#=returnType == null ? "int" : "ObjectResult<" +
MultiSchemaEscape(returnType, code) + ">"#> <#=code.Escape(edmFunction)#>(<#=code.StringAfter(paramList, ", ")#>MergeOption mergeOption)
        {
<#
                                            WriteFunctionParameters(parameters);
#>
            return base.<#=returnType == null ? "ExecuteFunction" : "ExecuteFunction<" + MultiSchemaEscape(returnType, code) + ">"#>("<#=edmFunction.Name#>",
mergeOption<#=code.StringBefore(", ", string.Join(", ", parameters.Select(p => p.ExecuteParameterName).ToArray()))#>);
        }
<#
                                        }
                                    }
                                }
                                region.End();
#>
```

```
}
<#
            }
            region.End();

            ////////
            ////////  Write EntityType classes.
            ////////
            region.Begin(CodeGenerationTools.GetResourceString("Template_RegionEntities"));
            foreach (EntityType entity in GetSourceSchemaTypes<EntityType>().OrderBy(e => e.Name))
            {
#>

/// <summary>
/// <#=SummaryComment(entity)#>
/// </summary><#=LongDescriptionCommentElement(entity, region.CurrentIndentLevel)#>
[EdmEntityTypeAttribute(NamespaceName="<#=entity.NamespaceName#>", Name="<#=entity.Name#>")]
[Serializable()]
[DataContractAttribute(IsReference=true)]
<#
                        foreach (EntityType subType in ItemCollection.GetItems<EntityType>().Where(b => b.BaseType == entity))
                        {
#>
[KnownTypeAttribute(typeof(<#=MultiSchemaEscape(subType, code)#>))]
<#
                        }
#>
<#=Accessibility.ForType(entity)#> <#=code.SpaceAfter(code.AbstractOption(entity))#>partial class <#=code.Escape(entity)#> : <#=BaseTypeName(entity, code)#>
{
<#
                        if (!entity.Abstract)
                        {
                                WriteFactoryMethod(entity, code);
                        }

                        region.Begin(CodeGenerationTools.GetResourceString("Template_RegionSimpleProperties"));
                        foreach (EdmProperty property in entity.Properties.Where(p => p.DeclaringType == entity && p.TypeUsage.EdmType is SimpleType))
                        {
                                VerifyGetterAndSetterAccessibilityCompatability(property);
                                WriteSimpleTypeProperty(property, code);
                        }
```

```
                region.End();

                region.Begin(CodeGenerationTools.GetResourceString("Template_RegionComplexProperties"));
                foreach (EdmProperty property in entity.Properties.Where(p => p.DeclaringType == entity && p.TypeUsage.EdmType is ComplexType))
                {
                        VerifyGetterAndSetterAccessibilityCompatability(property);
                        WriteComplexTypeProperty(property, code);
                }
                region.End();

                region.Begin(CodeGenerationTools.GetResourceString("Template_RegionNavigationProperties"));
                foreach (NavigationProperty navProperty in entity.NavigationProperties.Where(n => n.DeclaringType == entity))
                {
                        VerifyGetterAndSetterAccessibilityCompatability(navProperty);
#>

        /// <summary>
        /// <#=SummaryComment(navProperty)#>
        /// </summary><#=LongDescriptionCommentElement(navProperty, region.CurrentIndentLevel)#>
        [XmlIgnoreAttribute()]
        [SoapIgnoreAttribute()]
        [DataMemberAttribute()]
        [EdmRelationshipNavigationPropertyAttribute("<#=navProperty.RelationshipType.NamespaceName#>", "<#=navProperty.RelationshipType.Name#>",
"<#=navProperty.ToEndMember.Name#>")]
<#
                        if (navProperty.ToEndMember.RelationshipMultiplicity == RelationshipMultiplicity.Many)
                        {
#>
        <#=code.SpaceAfter(NewModifier(navProperty))#><#=Accessibility.ForProperty(navProperty)#>
EntityCollection<<#=MultiSchemaEscape(navProperty.ToEndMember.GetEntityType(), code)#>> <#=code.Escape(navProperty)#>
        {
                <#=code.SpaceAfter(Accessibility.ForGetter(navProperty))#>get
                {
                        return ((IEntityWithRelationships)this).RelationshipManager.GetRelatedCollection<<#=MultiSchemaEscape(navProperty.ToEndMember.GetEntityType(),
code)#>>("<#=navProperty.RelationshipType.FullName#>", "<#=navProperty.ToEndMember.Name#>");
                }
                <#=code.SpaceAfter(Accessibility.ForSetter(navProperty))#>set
                {
                        if ((value != null))
                        {
```

```
((IEntityWithRelationships)this).RelationshipManager.InitializeRelatedCollection<<#=MultiSchemaEscape(navProperty.ToEndMember.GetEntityType(),
code)#>>("<#=navProperty.RelationshipType.FullName#>", "<#=navProperty.ToEndMember.Name#>", value);
                                }
                        }
                }
<#
                          }
                          else
                          {
#>
                <#=code.SpaceAfter(NewModifier(navProperty))#><#=Accessibility.ForProperty(navProperty)#> <#=MultiSchemaEscape(navProperty.ToEndMember.GetEntityType(),
code)#> <#=code.Escape(navProperty)#>
                {
                        <#=code.SpaceAfter(Accessibility.ForGetter(navProperty))#>get
                        {
                                return ((IEntityWithRelationships)this).RelationshipManager.GetRelatedReference<<#=MultiSchemaEscape(navProperty.ToEndMember.GetEntityType(),
code)#>>("<#=navProperty.RelationshipType.FullName#>", "<#=navProperty.ToEndMember.Name#>").Value;
                        }
                        <#=code.SpaceAfter(Accessibility.ForSetter(navProperty))#>set
                        {
                                ((IEntityWithRelationships)this).RelationshipManager.GetRelatedReference<<#=MultiSchemaEscape(navProperty.ToEndMember.GetEntityType(),
code)#>>("<#=navProperty.RelationshipType.FullName#>", "<#=navProperty.ToEndMember.Name#>").Value = value;
                        }
                }
<#
                        string refPropertyName = navProperty.Name + "Reference";
                        if (entity.Members.Any(m => m.Name == refPropertyName))
                        {
                          //  6017 is the same error number that EntityClassGenerator uses.
                                Errors.Add(new System.CodeDom.Compiler.CompilerError(SourceCsdlPath, -1, -1, "6017", String.Format(CultureInfo.CurrentCulture,
                                                CodeGenerationTools.GetResourceString("Template_ConflictingGeneratedNavPropName"),
                                                        navProperty.Name, entity.FullName, refPropertyName)));
                        }
#>
                /// <summary>
                /// <#=SummaryComment(navProperty)#>
                /// </summary><#=LongDescriptionCommentElement(navProperty, region.CurrentIndentLevel)#>
                [BrowsableAttribute(false)]
                [DataMemberAttribute()]
                <#=Accessibility.ForProperty(navProperty)#> EntityReference<<#=MultiSchemaEscape(navProperty.ToEndMember.GetEntityType(), code)#>> <#=refPropertyName#>
                {
```

```
                        <#=code.SpaceAfter(Accessibility.ForGetter(navProperty))#>get
                        {
                                return ((IEntityWithRelationships)this).RelationshipManager.GetRelatedReference<<#=MultiSchemaEscape(navProperty.ToEndMember.GetEntityType(),
code)#>>("<#=navProperty.RelationshipType.FullName#>", "<#=navProperty.ToEndMember.Name#>");
                        }
                        <#=code.SpaceAfter(Accessibility.ForSetter(navProperty))#>set
                        {
                                if ((value != null))
                                {

((IEntityWithRelationships)this).RelationshipManager.InitializeRelatedReference<<#=MultiSchemaEscape(navProperty.ToEndMember.GetEntityType(),
code)#>>("<#=navProperty.RelationshipType.FullName#>", "<#=navProperty.ToEndMember.Name#>", value);
                                }
                        }
                }
<#
                                }
                        }
                        region.End();
#>
}
<#
                }
                region.End();

                ////////
                ////////  Write ComplexType classes.
                ////////
                region.Begin(CodeGenerationTools.GetResourceString("Template_RegionComplexTypes"));
                foreach (ComplexType complex in GetSourceSchemaTypes<ComplexType>().OrderBy(c => c.Name))
                {
#>

/// <summary>
/// <#=SummaryComment(complex)#>
/// </summary><#=LongDescriptionCommentElement(complex, region.CurrentIndentLevel)#>
[EdmComplexTypeAttribute(NamespaceName="<#=complex.NamespaceName#>", Name="<#=complex.Name#>")]
[DataContractAttribute(IsReference=true)]
[Serializable()]
<#=Accessibility.ForType(complex)#> partial class <#=code.Escape(complex)#> : ComplexObject
{
```

```
<#
                        WriteFactoryMethod(complex, code);
                        region.Begin(CodeGenerationTools.GetResourceString("Template_RegionSimpleProperties"));
                        foreach (EdmProperty property in complex.Properties.Where(p => p.DeclaringType == complex && p.TypeUsage.EdmType is SimpleType))
                        {
                                VerifyGetterAndSetterAccessibilityCompatability(property);
                                WriteSimpleTypeProperty(property, code);
                        }
                        region.End();

                        region.Begin(CodeGenerationTools.GetResourceString("Template_RegionComplexProperties"));
                        foreach (EdmProperty property in complex.Properties.Where(p => p.DeclaringType == complex && p.TypeUsage.EdmType is ComplexType))
                        {
                                VerifyGetterAndSetterAccessibilityCompatability(property);
                                WriteComplexTypeProperty(property, code);
                        }
                        region.End();
#>
}
<#
            }
            region.End();

            ////////
            ////////  Write enums.
            ////////
            region.Begin(CodeGenerationTools.GetResourceString("Template_RegionEnumTypes"));
            foreach (EnumType enumType in GetSourceSchemaTypes<EnumType>().OrderBy(c => c.Name))
            {
#>

/// <summary>
/// <#=SummaryComment(enumType)#>
/// </summary><#=LongDescriptionCommentElement(enumType, region.CurrentIndentLevel)#>
[EdmEnumTypeAttribute(NamespaceName="<#=enumType.NamespaceName#>", Name="<#=enumType.Name#>")]
[DataContractAttribute()]
<#
                        if (enumType.IsFlags)
                        {
#>
[FlagsAttribute()]
```

```
<#
                        }
#>
<#=Accessibility.ForType(enumType)#> enum <#=code.Escape(enumType)#> : <#=code.Escape(enumType.UnderlyingType.ClrEquivalentType, fullyQualifySystemTypes: false)#>
{
<#
                        foreach (EnumMember member in enumType.Members)
                        {
#>
            /// <summary>
            /// <#=SummaryComment(member)#>
            /// </summary><#=LongDescriptionCommentElement(member, region.CurrentIndentLevel + 1)#>
            [EnumMemberAttribute()]
            <#=code.Escape(member)#> = <#=member.Value#>,

<#
                        }

                        //  Remove the last comma and line break
                        if (enumType.Members.Any())
                        {
                                this.GenerationEnvironment.Remove(
                                        this.GenerationEnvironment.Length - (CurrentIndent.Length + 5), CurrentIndent.Length + 3);
                        }
#>
}
<#
            }
            region.End();

            if (!String.IsNullOrEmpty(namespaceName))
            {
                        PopIndent();
#>
}
<#
            }

            if (!VerifyTypeUniqueness(GetSourceSchemaTypes<GlobalItem>()
                                                                                                .Where(i => i is StructuralType || i is EnumType || i is EntityContainer)
                                                                                                .Select(i => code.GetGlobalItemName(i))))
```

```
        {
                return string.Empty;
        }
#>
<#+

        ////////
        ////////  Reusable Template Sections
        ////////

        ////////
        ////////  Write Factory Method.
        ////////
        private void WriteFactoryMethod(StructuralType structuralType, CodeGenerationTools code)
        {
                CodeRegion region = new CodeRegion(this, 1);

                string methodName = "Create" + structuralType.Name;
                UniqueIdentifierService uniqueIdentifier = new UniqueIdentifierService();
                string instanceName = code.Escape(uniqueIdentifier.AdjustIdentifier((code.CamelCase(structuralType.Name))));
                IEnumerable<FactoryMethodParameter> parameters = FactoryMethodParameter.CreateParameters(structuralType.Members.OfType<EdmProperty>().Where(p
=> IncludePropertyInFactoryMethod(structuralType, p)), uniqueIdentifier, MultiSchemaEscape, code);

                if (parameters.Count() == 0)
                        return;

                if (structuralType.Members.Any(m => m.Name == methodName))
                {
                        //  6029 is the same error number that EntityClassGenerator uses for this conflict.
                        Errors.Add(new System.CodeDom.Compiler.CompilerError(SourceCsdlPath, -1, -1, "6029",
                                String.Format(CultureInfo.CurrentCulture,
                                                CodeGenerationTools.GetResourceString("Template_FactoryMethodNameConflict"), methodName,
structuralType.FullName)));
                }

                region.Begin(CodeGenerationTools.GetResourceString("Template_RegionFactoryMethod"));
#>

        /// <summary>
        /// <#=String.Format(CultureInfo.CurrentCulture, CodeGenerationTools.GetResourceString("Template_FactoryMethodComment"), structuralType.Name)#>
        /// </summary><#=ParameterComments(parameters.Select(p => new Tuple<string, string>(p.RawParameterName, p.ParameterComment)), region.CurrentIndentLevel)#>
```

```
        public static <#=code.Escape(structuralType)#> <#=methodName#>(<#=string.Join(", ", parameters.Select(p => p.ParameterType + " " + p.ParameterName).ToArray())#>)
        {
            <#=code.Escape(structuralType)#> <#=instanceName#> = new <#=code.Escape(structuralType)#>();
<#+
            foreach (FactoryMethodParameter parameter in parameters)
            {
                if (parameter.IsComplexType)
                {
                    //  ComplexType initialization.
#>
            <#=instanceName#>.<#=code.Escape(parameter.Source)#> = StructuralObject.VerifyComplexObjectIsNotNull(<#=parameter.ParameterName#>,
"<#=parameter.Source.Name#>");
<#+
                }
                else
                {
                    //  SimpleType initialization.
#>
            <#=instanceName#>.<#=code.Escape(parameter.Source)#> = <#=parameter.ParameterName#>;
<#+
                }
            }
#>
            return <#=instanceName#>;
        }
<#+
            region.End();
        }


        ////////
        ////////  Write SimpleType Properties.
        ////////
        private void WriteSimpleTypeProperty(EdmProperty simpleProperty, CodeGenerationTools code)
        {
            MetadataTools ef = new MetadataTools(this);
#>

        /// <summary>
        /// <#=SummaryComment(simpleProperty)#>
        /// </summary><#=LongDescriptionCommentElement(simpleProperty, 1)#>
```

```
        [EdmScalarPropertyAttribute(EntityKeyProperty=<#=code.CreateLiteral(ef.IsKey(simpleProperty))#>, IsNullable=<#=code.CreateLiteral(ef.IsNullable(simpleProperty))#>)]
        [DataMemberAttribute()]
        <#=code.SpaceAfter(NewModifier(simpleProperty))#><#=Accessibility.ForProperty(simpleProperty)#> <#=MultiSchemaEscape(simpleProperty.TypeUsage, code)#>
<#=code.Escape(simpleProperty)#>
        {
                <#=code.SpaceAfter(Accessibility.ForGetter(simpleProperty))#>get
                {
<#+             if (ef.ClrType(simpleProperty.TypeUsage) == typeof(byte[]))
                        {
#>
                        return StructuralObject.GetValidValue(<#=code.FieldName(simpleProperty)#>);
<#+
                        }
                        else
                        {
#>
                        return <#=code.FieldName(simpleProperty)#>;
<#+
                        }
#>
                }
                <#=code.SpaceAfter(Accessibility.ForSetter((simpleProperty)))#>set
                {
<#+
                if (ef.IsKey(simpleProperty))
                        {
                                if (ef.ClrType(simpleProperty.TypeUsage) == typeof(byte[]))
                                {
#>
                        if (!StructuralObject.BinaryEquals(<#=code.FieldName(simpleProperty)#>, value))
<#+
                                }
                                else
                                {
#>
                        if (<#=code.FieldName(simpleProperty)#> != value)
<#+
                                }
#>
                        {
<#+
```

```
                        PushIndent(CodeRegion.GetIndent(1));
                                }
#>
                                <#=ChangingMethodName(simpleProperty)#>(value);
                                ReportPropertyChanging("<#=simpleProperty.Name#>");
                                <#=code.FieldName(simpleProperty)#> = <#=CastToEnumType(simpleProperty.TypeUsage,
code)#>StructuralObject.SetValidValue(<#=CastToUnderlyingType(simpleProperty.TypeUsage, code)#>value<#=OptionalNullableParameterForSetValidValue(simpleProperty, code)#>,
"<#=simpleProperty.Name#>");
                                ReportPropertyChanged("<#=simpleProperty.Name#>");
                                <#=ChangedMethodName(simpleProperty)#>();
<#+
                        if (ef.IsKey(simpleProperty))
                                {
                        PopIndent();
#>
                                }
<#+
                                }
#>
                        }
                }
                private <#=MultiSchemaEscape(simpleProperty.TypeUsage, code)#> <#=code.FieldName(simpleProperty)#><#=code.StringBefore(" = ",
code.CreateLiteral(simpleProperty.DefaultValue))#>;
                partial void <#=ChangingMethodName(simpleProperty)#>(<#=MultiSchemaEscape(simpleProperty.TypeUsage, code)#> value);
                partial void <#=ChangedMethodName(simpleProperty)#>();
<#+
                }

                ////////
                ////////  Write ComplexType Properties.
                ////////
                private void WriteComplexTypeProperty(EdmProperty complexProperty, CodeGenerationTools code)
                {
#>

                /// <summary>
                /// <#=SummaryComment(complexProperty)#>
                /// </summary><#=LongDescriptionCommentElement(complexProperty, 1)#>
                [EdmComplexPropertyAttribute()]
                [DesignerSerializationVisibility(DesignerSerializationVisibility.Content)]
                [XmlElement(IsNullable=true)]
```

```
        [SoapElement(IsNullable=true)]
        [DataMemberAttribute()]
        <#=code.SpaceAfter(NewModifier(complexProperty))#><#=Accessibility.ForProperty(complexProperty)#> <#=MultiSchemaEscape(complexProperty.TypeUsage, code)#>
<#=code.Escape(complexProperty)#>
        {
                <#=code.SpaceAfter(Accessibility.ForGetter(complexProperty))#>get
                {
                        <#=code.FieldName(complexProperty)#> = GetValidValue(<#=code.FieldName(complexProperty)#>, "<#=complexProperty.Name#>", false,
<#=InitializedTrackingField(complexProperty, code)#>);
                        <#=InitializedTrackingField(complexProperty, code)#> = true;
                        return <#=code.FieldName(complexProperty)#>;
                }
                <#=code.SpaceAfter(Accessibility.ForSetter(complexProperty))#>set
                {
                        <#=ChangingMethodName(complexProperty)#>(value);
                        ReportPropertyChanging("<#=complexProperty.Name#>");
                        <#=code.FieldName(complexProperty)#> = SetValidValue(<#=code.FieldName(complexProperty)#>, value);
                        <#=InitializedTrackingField(complexProperty, code)#> = true;
                        ReportPropertyChanged("<#=complexProperty.Name#>");
                        <#=ChangedMethodName(complexProperty)#>();
                }
        }
        private <#=MultiSchemaEscape(complexProperty.TypeUsage, code)#> <#=code.FieldName(complexProperty)#>;
        private bool <#=InitializedTrackingField(complexProperty, code)#>;
        partial void <#=ChangingMethodName(complexProperty)#>(<#=MultiSchemaEscape(complexProperty.TypeUsage, code)#> value);
        partial void <#=ChangedMethodName(complexProperty)#>();
<#+
        }

        private void WriteFunctionParameters(IEnumerable<FunctionImportParameter> parameters)
        {
                foreach (FunctionImportParameter parameter in parameters)
                {
                        if (!parameter.NeedsLocalVariable)
                        {
                                continue;
                        }
#>
                ObjectParameter <#=parameter.LocalVariableName#>;
                if (<#=parameter.IsNullableOfT ? parameter.FunctionParameterName + ".HasValue" : parameter.FunctionParameterName + " != null"#>)
                {
```

```
                        <#=parameter.LocalVariableName#> = new ObjectParameter("<#=parameter.EsqlParameterName#>", <#=parameter.FunctionParameterName#>);
                }
                else
                {
                        <#=parameter.LocalVariableName#> = new ObjectParameter("<#=parameter.EsqlParameterName#>", typeof(<#=parameter.RawClrTypeName#>));
                }

<#+
                }
        }

        private void WriteLazyLoadingEnabled(EntityContainer container)
        {
                string lazyLoadingAttributeValue = null;
                string lazyLoadingAttributeName = MetadataConstants.EDM_ANNOTATION_09_02 + ":LazyLoadingEnabled";
                if(MetadataTools.TryGetStringMetadataPropertySetting(container, lazyLoadingAttributeName, out lazyLoadingAttributeValue))
                {
                        bool isLazyLoading = false;
                        if(bool.TryParse(lazyLoadingAttributeValue, out isLazyLoading))
                        {
#>
                this.ContextOptions.LazyLoadingEnabled = <#=isLazyLoading.ToString().ToLowerInvariant()#>;
<#+
                        }
                }
        }

////////
////////  Declare Template Public Properties.
////////
public string SourceCsdlPath{ get; set; }
public string ModelNamespace{ get; set; }
public EdmItemCollection ItemCollection{ get; set; }
public IEnumerable<string> ReferenceCsdlPaths{ get; set; }
public Nullable<bool> CreateContextAddToMethods{ get; set; }
public Dictionary<string, string> EdmToObjectNamespaceMap
{
        get { return _edmToObjectNamespaceMap; }
        set { _edmToObjectNamespaceMap = value; }
}
public Dictionary<string, string> _edmToObjectNamespaceMap = new Dictionary<string, string>();
```

```
public Double SourceEdmVersion
{
	get
	{
		if (ItemCollection != null)
		{
			return ItemCollection.EdmVersion;
		}

		return 0.0;
	}
}

private bool VerifyTypeUniqueness(IEnumerable<string> types)
{
	HashSet<string> hash = new HashSet<string>();
	foreach (string type in types)
	{
		if (!hash.Add(type))
		{
			//  6034 is the error number used by System.Data.Entity.Design EntityClassGenerator.
			Errors.Add(new System.CodeDom.Compiler.CompilerError(SourceCsdlPath, -1, -1, "6034",
				String.Format(CultureInfo.CurrentCulture,
					CodeGenerationTools.GetResourceString("Template_DuplicateTopLevelType"),
					type)));
			return false;
		}
	}
	return true;
}

void ApplyUserSettings(UserSettings userSettings)
{
	//  Setup template UserSettings.
	if (SourceCsdlPath == null)
	{
#if !PREPROCESSED_TEMPLATE
		if(userSettings.SourceCsdlPath == "$" + "edmxInputFile" + "$")
		{
			Errors.Add(new System.CodeDom.Compiler.CompilerError(Host.TemplateFile, -1, -1, "",
				CodeGenerationTools.GetResourceString("Template_ReplaceVsItemTemplateToken")));
```

```
                        return;
                    }

                    SourceCsdlPath = Host.ResolvePath(userSettings.SourceCsdlPath);
#else
                    SourceCsdlPath = userSettings.SourceCsdlPath;
#endif
        }

        // normalize the path, remove ..\ from it
        SourceCsdlPath = Path.GetFullPath(SourceCsdlPath);


        if (ReferenceCsdlPaths == null)
        {
            ReferenceCsdlPaths = userSettings.ReferenceCsdlPaths;
        }

        if (!CreateContextAddToMethods.HasValue)
        {
            CreateContextAddToMethods = userSettings.CreateContextAddToMethods;
        }

        DefaultSummaryComment = CodeGenerationTools.GetResourceString("Template_CommentNoDocumentation");
}

class UserSettings
{
        public string SourceCsdlPath{ get; set; }
        public string[] ReferenceCsdlPaths{ get; set; }
        public bool FullyQualifySystemTypes{ get; set; }
        public bool CreateContextAddToMethods{ get; set; }
        public bool CamelCaseFields{ get; set; }
}

string GetNamespaceName(CodeGenerationTools code)
{
        string namespaceName = code.VsNamespaceSuggestion();

#if PREPROCESSED_TEMPLATE
        if (String.IsNullOrEmpty(namespaceName))
```

```
        {
            namespaceName = GetObjectNamespace(ModelNamespace);
        }
#endif
        return namespaceName;
}

string MultiSchemaEscape(TypeUsage usage, CodeGenerationTools code)
{
        MetadataTools ef = new MetadataTools(this);
        if (usage.EdmType != null)
        {
            if (usage.EdmType is StructuralType)
            {
                return MultiSchemaEscape(usage.EdmType, code);
            }
            else if (usage.EdmType is EnumType)
            {
                string typeName = MultiSchemaEscape(usage.EdmType, code);
                if (ef.IsNullable(usage))
                {
                    return String.Format(CultureInfo.InvariantCulture, "Nullable<{0}>", typeName);
                }

                return typeName;
            }
        }

        return code.Escape(usage);
}

string MultiSchemaEscape(EdmType type, CodeGenerationTools code)
{
        if (type.NamespaceName != ModelNamespace)
        {
            return code.CreateFullName(code.EscapeNamespace(GetObjectNamespace(type.NamespaceName)), code.Escape(type));
        }

        return code.Escape(type);
}
```

```
string NewModifier(NavigationProperty navigationProperty)
{
	Type baseType = typeof(EntityObject);
	return NewModifier(baseType, navigationProperty.Name);
}

string NewModifier(EdmFunction edmFunction)
{
	Type baseType = typeof(ObjectContext);
	return NewModifier(baseType, edmFunction.Name);
}

string NewModifier(EntitySet set)
{
	Type baseType = typeof(ObjectContext);
	return NewModifier(baseType, set.Name);
}

string NewModifier(EdmProperty property)
{
	Type baseType;
	if (property.DeclaringType.BuiltInTypeKind == BuiltInTypeKind.EntityType)
	{
		baseType = typeof(EntityObject);
	}
	else
	{
		baseType = typeof(ComplexObject);
	}
	return NewModifier(baseType, property.Name);
}

string NewModifier(Type type, string memberName)
{
	if (HasBaseMemberWithMatchingName(type, memberName))
	{
		return "new";
	}
	return string.Empty;
}
```

```
static bool HasBaseMemberWithMatchingName(Type type, string memberName)
{
	BindingFlags bindingFlags = BindingFlags.FlattenHierarchy | BindingFlags.NonPublic | BindingFlags.Public
					| BindingFlags.Instance | BindingFlags.Static;
	return type.GetMembers(bindingFlags).Where(m => IsVisibleMember(m)).Any(m => m.Name == memberName);
}

string CastToEnumType(TypeUsage typeUsage, CodeGenerationTools code)
{
	EnumType type = typeUsage.EdmType as EnumType;
	if (type == null)
		return string.Empty;

	return "(" + MultiSchemaEscape(typeUsage, code) + ")";
}

string CastToUnderlyingType(TypeUsage typeUsage, CodeGenerationTools code)
{
	MetadataTools ef = new MetadataTools(this);
	EnumType type = typeUsage.EdmType as EnumType;
	if (type == null)
	{
		return string.Empty;
	}

	string clrType = code.Escape(type.UnderlyingType.ClrEquivalentType, fullyQualifySystemTypes: false);

	if (ef.IsNullable(typeUsage))
	{
		clrType = String.Format(CultureInfo.InvariantCulture, "Nullable<{0}>", clrType);
	}

	return "(" + clrType + ")";
}

string ChangingMethodName(EdmMember member)
{
	return string.Format(CultureInfo.InvariantCulture, "On{0}Changing", member.Name);
}

string ChangedMethodName(EdmMember member)
```

```
{
	return string.Format(CultureInfo.InvariantCulture, "On{0}Changed", member.Name);
}

string InitializedTrackingField(EdmProperty property, CodeGenerationTools code)
{
	string namePart = property.Name + "Initialized";
	if (code.CamelCaseFields)
	{
		namePart = code.CamelCase(namePart);
	}
	return "_" + namePart;
}

string OptionalNullableParameterForSetValidValue(EdmMember member, CodeGenerationTools code)
{
	MetadataTools ef = new MetadataTools(this);
	string list = string.Empty;
	PrimitiveType type = member.TypeUsage.EdmType as PrimitiveType;
	if (type != null && type.ClrEquivalentType.IsClass)
	{
		MetadataProperty storeGeneratedPatternProperty = null;
		bool isNullable = ef.IsNullable(member.TypeUsage) ||
			(member.MetadataProperties.TryGetValue(MetadataConstants.EDM_ANNOTATION_09_02 + ":StoreGeneratedPattern", false, out
storeGeneratedPatternProperty) &&
			 Object.Equals(storeGeneratedPatternProperty.Value, "Computed"));
		list += ", " + code.CreateLiteral(isNullable);
	}
	return list;
}

static bool IsVisibleMember(MemberInfo memberInfo)
{
	if (memberInfo is EventInfo)
	{
		EventInfo ei = (EventInfo)memberInfo;
		MethodInfo add = ei.GetAddMethod();
		MethodInfo remove = ei.GetRemoveMethod();
		return IsVisibleMethod(add) || IsVisibleMethod(remove);
	}
	else if (memberInfo is FieldInfo)
```

```
        {
                FieldInfo fi = (FieldInfo)memberInfo;
                return !fi.IsPrivate && !fi.IsAssembly;
        }
        else if (memberInfo is MethodBase)
        {
                MethodBase mb = (MethodBase)memberInfo;
                if (mb.IsSpecialName)
                        return false;
                return IsVisibleMethod(mb);
        }
        else if (memberInfo is PropertyInfo)
        {
                PropertyInfo pi = (PropertyInfo)memberInfo;
                MethodInfo get = pi.GetGetMethod();
                MethodInfo set = pi.GetSetMethod();
                return IsVisibleMethod(get) || IsVisibleMethod(set);
        }

        return false;
}

static bool IsVisibleMethod(MethodBase methodBase)
{
        if (methodBase == null)
                return false;

        return !methodBase.IsPrivate && !methodBase.IsAssembly;
}

IEnumerable<T> GetSourceSchemaTypes<T>() where T : GlobalItem
{
        if (Path.GetExtension(SourceCsdlPath) != ".edmx")
        {
                return ItemCollection.GetItems<T>().Where(e => e.MetadataProperties.Any(mp => mp.Name == "SchemaSource" && (string)mp.Value == SourceCsdlPath));
        }
        else
        {
                return ItemCollection.GetItems<T>();
        }
}
```

```
string EndName(AssociationType association, int index)
{
	return association.AssociationEndMembers[index].Name;
}

string EndMultiplicity(AssociationType association, int index, CodeGenerationTools code)
{
	return code.CreateLiteral(association.AssociationEndMembers[index].RelationshipMultiplicity);
}

string EscapeEndTypeName(AssociationType association, int index, CodeGenerationTools code)
{
	EntityType entity = association.AssociationEndMembers[index].GetEntityType();
	return code.CreateFullName(code.EscapeNamespace(GetObjectNamespace(entity.NamespaceName)), code.Escape(entity));
}

string GetObjectNamespace(string csdlNamespaceName)
{
	string objectNamespace;
	if (EdmToObjectNamespaceMap.TryGetValue(csdlNamespaceName, out objectNamespace))
	{
		return objectNamespace;
	}

	return csdlNamespaceName;
}

void UpdateObjectNamespaceMap(string objectNamespace)
{
	if(objectNamespace != ModelNamespace && !EdmToObjectNamespaceMap.ContainsKey(ModelNamespace))
	{
		EdmToObjectNamespaceMap.Add(ModelNamespace, objectNamespace);
	}
}

static string FixParameterName(string name, CodeGenerationTools code)
{
	//  Change any property that is 'id' (case insensitive) to 'ID'
	//  since 'iD' is a violation of FxCop rules.
	if (StringComparer.OrdinalIgnoreCase.Equals(name, "id"))
```

```
        {
            //  Return all lower case since it is an abbreviation, not an acronym.
            return "id";
        }
        return code.CamelCase(name);
}

string BaseTypeName(EntityType entity, CodeGenerationTools code)
{
        return entity.BaseType == null ? "EntityObject" : MultiSchemaEscape(entity.BaseType, code);
}

bool IncludePropertyInFactoryMethod(StructuralType factoryType, EdmProperty edmProperty)
{
        if (edmProperty.Nullable)
        {
            return false;
        }

        if (edmProperty.DefaultValue != null)
        {
            return false;
        }

        if ((Accessibility.ForReadOnlyProperty(edmProperty) != "public" && Accessibility.ForWriteOnlyProperty(edmProperty) != "public") ||
            (factoryType != edmProperty.DeclaringType && Accessibility.ForWriteOnlyProperty(edmProperty) == "private")
          )
        {
            //  There is no public part to the property.
            return false;
        }

        return true;
}

class FactoryMethodParameter
{
        public EdmProperty Source;
        public string RawParameterName;
        public string ParameterName;
        public string ParameterType;
```

```
        public string ParameterComment;
        public bool IsComplexType;

        public static IEnumerable<FactoryMethodParameter> CreateParameters(IEnumerable<EdmProperty> properties, UniqueIdentifierService unique, Func<TypeUsage,
CodeGenerationTools, string> multiSchemaEscape, CodeGenerationTools code)
        {
                List<FactoryMethodParameter> parameters = new List<FactoryMethodParameter>();
                foreach (EdmProperty property in properties)
                {
                        FactoryMethodParameter parameter = new FactoryMethodParameter();
                        parameter.Source = property;
                        parameter.IsComplexType = property.TypeUsage.EdmType is ComplexType;
                        parameter.RawParameterName = unique.AdjustIdentifier(FixParameterName(property.Name, code));
                        parameter.ParameterName = code.Escape(parameter.RawParameterName);
                        parameter.ParameterType = multiSchemaEscape(property.TypeUsage, code);
                        parameter.ParameterComment = String.Format(CultureInfo.CurrentCulture,
CodeGenerationTools.GetResourceString("Template_CommentFactoryMethodParam"), property.Name);
                        parameters.Add(parameter);
                }

                return parameters;
        }
}

string DefaultSummaryComment{ get; set; }

string SummaryComment(MetadataItem item)
{
        if (item.Documentation != null && item.Documentation.Summary != null)
        {
                return PrefixLinesOfMultilineComment(XMLCOMMENT_START + " ", XmlEntityize(item.Documentation.Summary));
        }

        if (DefaultSummaryComment != null)
        {
                return DefaultSummaryComment;
        }

        return string.Empty;
}
```

```
string LongDescriptionCommentElement(MetadataItem item, int indentLevel)
{
	if (item.Documentation != null && !String.IsNullOrEmpty(item.Documentation.LongDescription))
	{
		string comment = Environment.NewLine;
		string lineStart = CodeRegion.GetIndent(indentLevel) + XMLCOMMENT_START + " ";
		comment += lineStart + "<LongDescription>" + Environment.NewLine;
		comment += lineStart + PrefixLinesOfMultilineComment(lineStart, XmlEntityize(item.Documentation.LongDescription)) + Environment.NewLine;
		comment += lineStart + "</LongDescription>";
		return comment;
	}
	return string.Empty;
}

string PrefixLinesOfMultilineComment(string prefix, string comment)
{
	return comment.Replace(Environment.NewLine, Environment.NewLine + prefix);
}

string ParameterComments(IEnumerable<Tuple<string, string>> parameters, int indentLevel)
{
	System.Text.StringBuilder builder = new System.Text.StringBuilder();
	foreach (Tuple<string, string> parameter in parameters)
	{
		builder.AppendLine();
		builder.Append(CodeRegion.GetIndent(indentLevel));
		builder.Append(XMLCOMMENT_START);
		builder.Append(String.Format(CultureInfo.InvariantCulture, " <param name=\"{0}\">{1}</param>", parameter.Item1, parameter.Item2));
	}
	return builder.ToString();
}

string XmlEntityize(string text)
{
	if (string.IsNullOrEmpty(text))
	{
		return string.Empty;
	}

	text = text.Replace("&","&amp;");
	text = text.Replace("<","&lt;").Replace(">","&gt;");
```

```
            string id = Guid.NewGuid().ToString();
            text = text.Replace(Environment.NewLine, id);
            text = text.Replace("\r", "&#xD;").Replace("\n","&#xA;");
            text = text.Replace(id, Environment.NewLine);
            return text.Replace("\'","&apos;").Replace("\"","&quot;");
}

const string XMLCOMMENT_START = "///";
IEnumerable<EdmProperty> GetProperties(StructuralType type)
{
            if (type.BuiltInTypeKind == BuiltInTypeKind.EntityType)
            {
                        return ((EntityType)type).Properties;
            }
            else
            {
                        return ((ComplexType)type).Properties;
            }
}

protected void VerifyGetterAndSetterAccessibilityCompatability(EdmMember member)
{
            string rawGetterAccessibility = Accessibility.ForReadOnlyProperty(member);
            string rawSetterAccessibility = Accessibility.ForWriteOnlyProperty(member);

            if ((rawGetterAccessibility == "internal" && rawSetterAccessibility ==   "protected") ||
                        (rawGetterAccessibility == "protected" && rawSetterAccessibility == "internal"))

            {
                          Errors.Add(new System.CodeDom.Compiler.CompilerError(SourceCsdlPath, -1, -1, "6033", String.Format(CultureInfo.CurrentCulture,
                                      CodeGenerationTools.GetResourceString("GeneratedPropertyAccessibilityConflict"),
                                                  member.Name, rawGetterAccessibility, rawSetterAccessibility)));
            }
}

private void VerifyEntityTypeAndSetAccessibilityCompatability(EntitySet set)
{
            string typeAccess = Accessibility.ForType(set.ElementType);
            string setAccess = Accessibility.ForReadOnlyProperty(set);

            if(typeAccess == "internal" && (setAccess == "public" || setAccess == "protected"))
```

```
            {
              Errors.Add(new System.CodeDom.Compiler.CompilerError(SourceCsdlPath, -1, -1, "6036", String.Format(CultureInfo.CurrentCulture,
                                    CodeGenerationTools.GetResourceString("EntityTypeAndSetAccessibilityConflict"),
                                                set.ElementType.Name, typeAccess, set.Name, setAccess)));
            }
}

private void DefineMetadata()
{
            TemplateMetadata[MetadataConstants.TT_TEMPLATE_NAME] = "CSharpCodeGen";
            TemplateMetadata[MetadataConstants.TT_TEMPLATE_VERSION] = "5.0";
            TemplateMetadata[MetadataConstants.TT_MINIMUM_ENTITY_FRAMEWORK_VERSION] = "4.0";
}

sealed class UniqueIdentifierService
{
            private readonly HashSet<string> _knownIdentifiers;

            public UniqueIdentifierService()
            {
                        _knownIdentifiers = new HashSet<string>(StringComparer.Ordinal);
            }

            /// <summary>
            ///  Makes the supplied identifier  unique within the scope by adding
            ///  a suffix (1, 2, 3, ...), and returns the unique identifier.
            /// </summary>
            public string AdjustIdentifier(string identifier)
            {
                        //  find a unique name by adding suffix as necessary
                        var numberOfConflicts = 0;
                        var adjustedIdentifier = identifier;

                        while (!_knownIdentifiers.Add(adjustedIdentifier))
                        {
                                    ++numberOfConflicts;
                                    adjustedIdentifier = identifier + numberOfConflicts.ToString(CultureInfo.InvariantCulture);
                        }

                        return adjustedIdentifier;
            }
```

```
}

#>
```