Licenciatura En Gestión Tecnológica

Programación Avanzada II

Formación Continua

Entity Framework

Ing. Mariano Juiz

Agenda

- Introducción
- Prerequisitos
- Arquitectura.
- Enfoques.
- Creación de un EDM.
- LINQ.
- LINQ To Entities.
- Lazy loading, Eager Loading.
- Ciclo de vida.
- Operaciones CRUD.
- Tipos de Relaciones.

Introducción

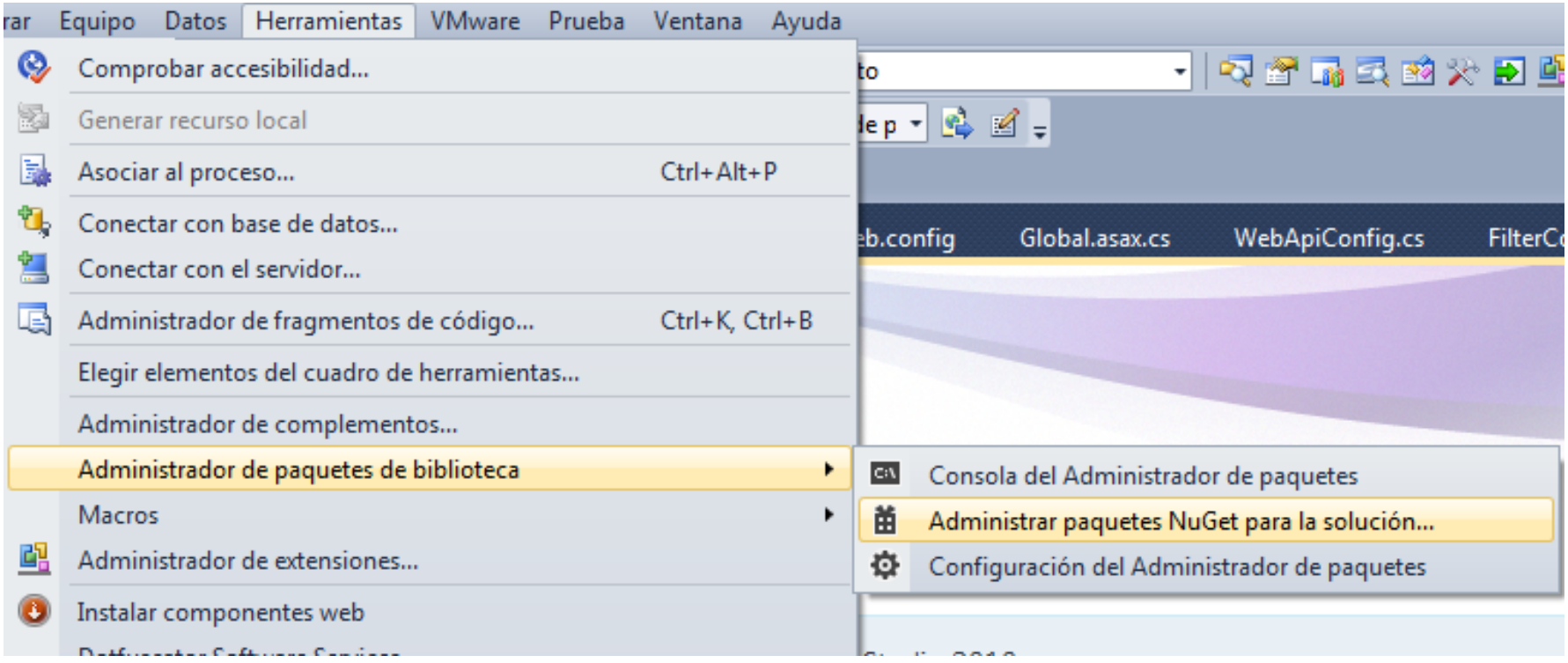
El **mapeo objeto-relacional** (más conocido por su nombre en inglés, Object-Relational mapping, o sus siglas O/RM, ORM, y O/R mapping) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional, utilizando un motor de persistencia.

Un ORM aporta ayuda en la resolución del denominado **desajuste de impedancia** (o impedance mismatch en inglés).
El desajuste de impedancia refiere a las **diferencias existentes** entre los **modelos relacionales** (base de datos) y los **modelos conceptuales** implementados en lenguajes de programación orientados a objetos.

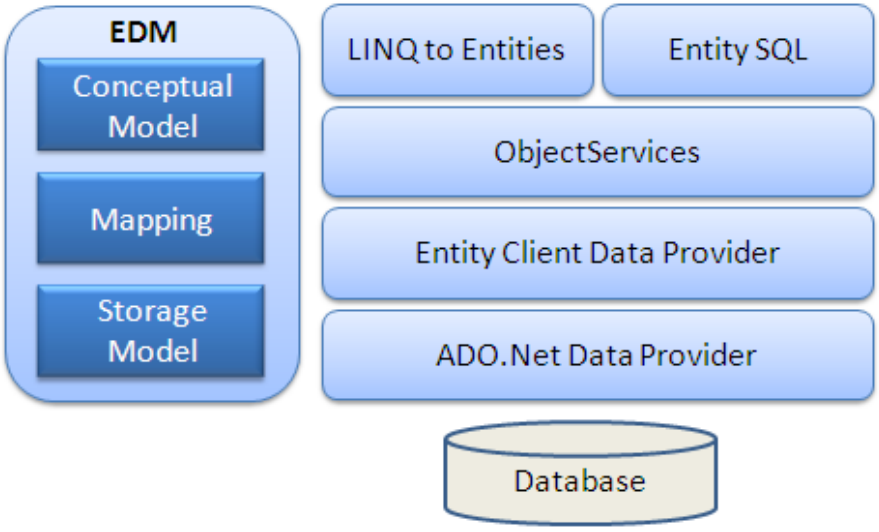
ADO.NET Entity Framework es un ORM que permite a los desarrolladores crear aplicaciones de acceso a datos programando con un modelo de aplicaciones conceptuales en lugar de programar directamente con un esquema de almacenamiento relacional. El objetivo es reducir la cantidad de código y el mantenimiento necesarios para las aplicaciones orientadas a datos.

Prerequisitos

- .NET Framework 3.5 o superior
- Visual Studio 2008 o superior
- ADO.Net Entity Framework 4.0 o superior
- SQL Server Express or SQL Server 2005 o superior



Arquitectura



EDM: EDM (Entity Data Model) es la definición del mapeo que creamos entre la base de datos y nuestro modelo conceptual o de entidades.

Un modelo se guarda en un fichero con extensión *edmx,* que a través del lenguaje XML declara 3 secciones (que aunque puede estar en ficheros distintos, la norma es que estén en un solo fichero):

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="2.0" xmlns:edmx="http://schemas.microsoft.com/ado/2008/10/edmx">
  <!-- EF Runtime content -->
  <edmx:Runtime>
    <!-- SSDL content -->
    <edmx:StorageModels>
    <!-- CSDL content -->
    <edmx:ConceptualModels>
    <!-- C-S mapping content -->
    <edmx:Mappings>
  </edmx:Runtime>
```

Enfoques

Existen 3 formas o enfoques distintos para utilizar EF:

- **Data Base First (Este será el enfoque que seguiremos)**

  A partir de un modelo relacional genero el EDM. Se trabaja directamente con **DbContext**.

- **Model First**

  A partir del modelo conceptual (EDM) genero el modelo relacional. Se trabaja directamente con **DbContext**.

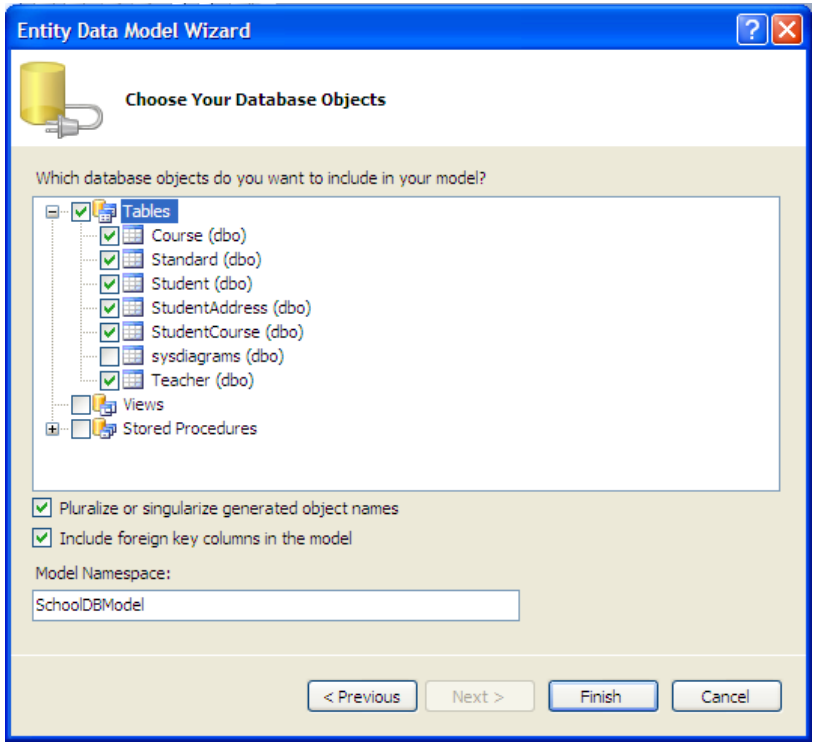
- **Code First:**

  No se trabaja con el EDMX designer. Se parte de un modelo puro de entidades en programación orientada a objetos (clases POCO) y a partir de ahí se genera el modelo de base de datos relacional. Incorpora dos nuevos tipos: **DbContext,** que es una alternativa simplificada de DbContext, y DBset en vez de **ObjectSet(Of TEntity). Solo Disponible a partir de Entity Framework 4.1**

Enfoque DataBase First

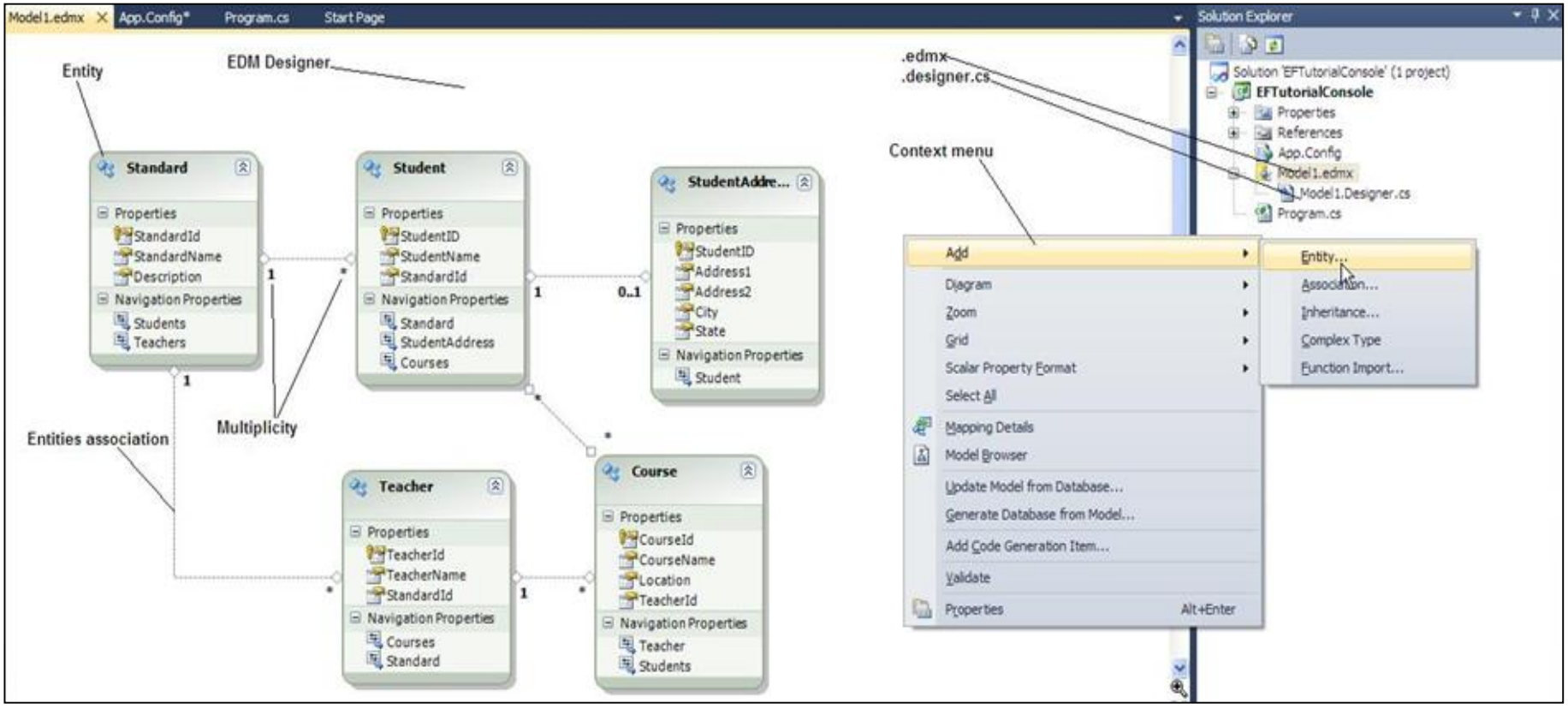
Pasos para crear el EDMX:

1. Crear un proyecto Sitio Web o Aplicación Web o Consola
2. Botón derecho sobre el proyecto, elegir “Agregar Nuevo Item” y escoger el template ADO.NET Entity Data Model
3. En el Data Model wizard, elegir la opción **Generar desde la Base de Datos.**
4. Elegir una conexión de base de datos.
5. Elegir los objetos de base de datos que se necesitan mapear



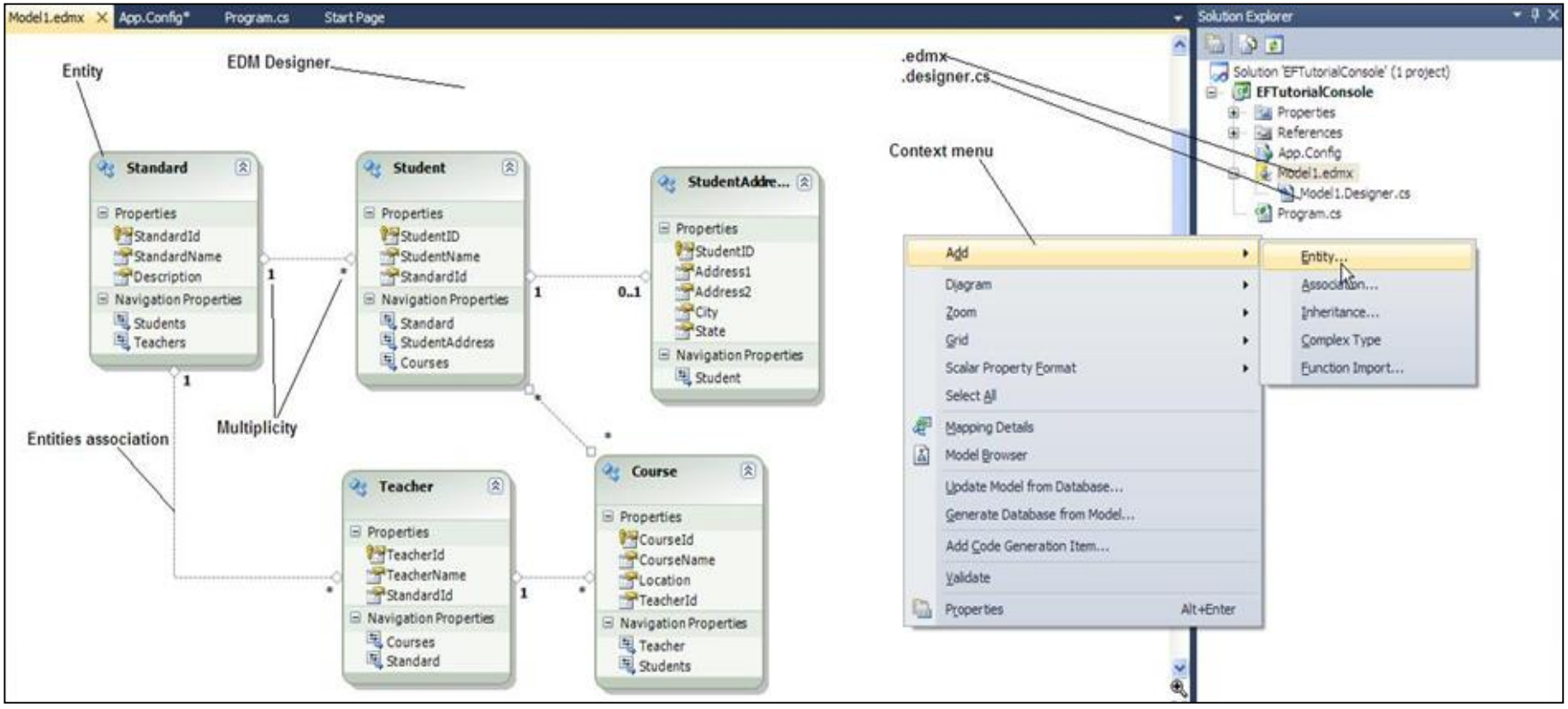
Enfoque DataBase First

6. Finalizado 5, se genera el EDMX:



Enfoque DataBase First

6. Finalizado 5, se genera el EDMX:



Enfoque Code First

1. Crear la clase para el Modelo

```
namespace EFEjemplo.Modelo {
  public class Persona
  {
     public int Id { get; set; }
     public string Apellido { get; set; }
     public string Nombre { get; set; }
  }
}
```

2. Crear la clase para Administrar la relación entre la Base de Datos y el DBContext

```
using System.Data.Entity;
namespace EFEjemplo.Modelo
{
public class MiContexto : DbContext
   {
        public MiContexto() : base("name=CONEXION_WEBCONFIG") {}
        public DbSet<Persona> Personas{ get; set; }
   }
}
```

Enfoque Code First

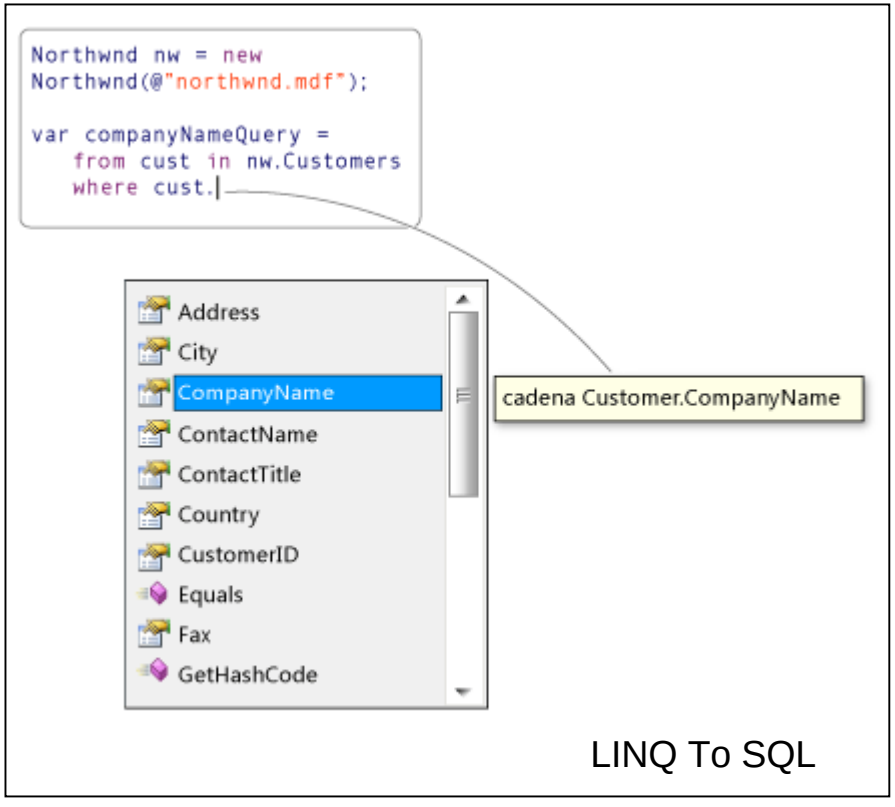
3. Crear el controlador para llamar a la clase administradora

```
using System.Linq;
using System.Web.Mvc;
using EFEjemplo.Modelo;
namespace EFEjemplo.Controladores
{
    public class PersonaController : Controller
    {
        MiContexto contexto;
        public PersonaController()
        {
            contexto = new MiContexto();
        }
        [HttpPost]
        public ActionResult Create(Persona persona)
        {
            contexto.Personas.Add(persona);
            contexto.SaveChanges();
            return RedirectToAction("Index");
        }
}
```

LINQ

Language-Integrated Query (LINQ) es una innovación introducida en Visual Studio 2008 y .NET Framework versión 3.5 que elimina la distancia que separa el mundo de los objetos y el mundo de los datos.

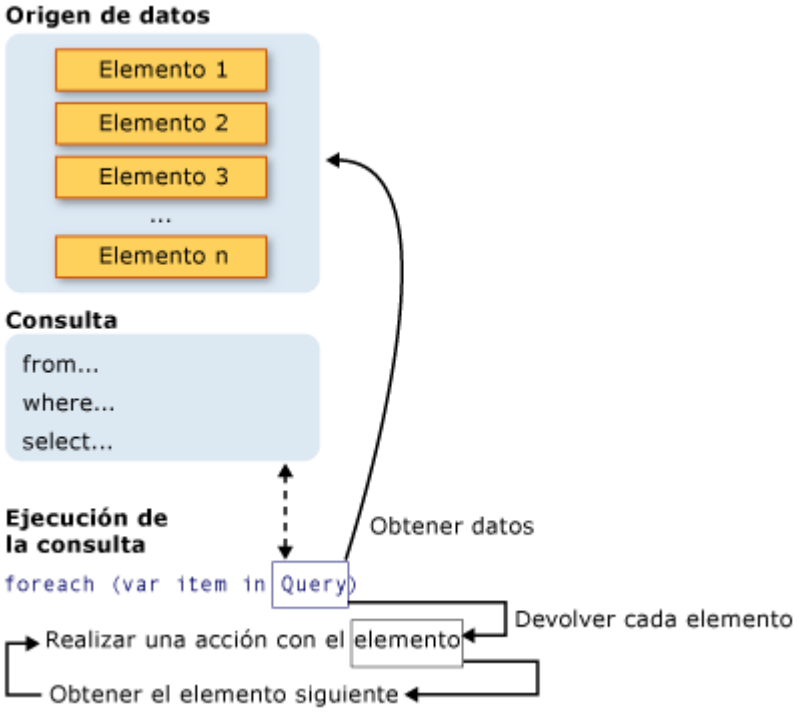
Tradicionalmente, las consultas con datos se expresan como cadenas sencillas, sin comprobación de tipos en tiempo de compilación ni compatibilidad con IntelliSense. Además, es necesario aprender un lenguaje de consultas diferente para cada tipo de origen de datos: bases de datos SQL, documentos XML, etc. LINQ Permite escribir estas consultas directamente en C#. Las consultas se escriben para colecciones de objetos fuertemente tipadas, utilizando palabras clave del lenguaje y operadores con los que se está familiarizado.



LINQ (2)

Las tres partes de una consulta en LINQ

```
//  1. Origen de datos.
int[] numeros = new int[] { 0, 1, 2, 3, 4, 5, 6 };


// 2. Creación de la consulta.
// numQuery es un IEnumerable<int>
var numQuery =
    from num in numeros
    where (num % 2) == 0
    select num;


// 3. Ejecución de la consulta.
foreach (int num in numQuery)
{
    Response.Write(string.Format("{0}", num));
}
```



LINQ To…

LINQ se puede usar con distintos orígenes de datos:

LINQ To SQL: Bases de datos SQL Server

LINQ To Objects: Cadenas, numeros, objetos, etc

LINQ To XML: Documentos XML

LINQ To Datasets: Conjunto de datos de ADO.NET

LINQ To Entities (1)

LINQ se usa en este caso para escribir consultas hacia el EDM. Devuelve las entidades definidas en el modelo conceptual

En este caso usamos la **sintaxis de expresiones de consulta:**

```
private void ListarProductos(int idCliente)
{
    //1) Origen de Datos
    EFPW3 context = new EFPW3();

    //2) Consulta: Sintaxis de consulta
    var productos = from p in context.Producto
                    where p.Cliente.Id == idCliente
                    select p;

    //3) Ejecución de Consulta
    foreach (Producto p in productos)
    {
        lblMensaje.Text += p.Nombre + " - ";
    }
}
```

LINQ To Entities (2)

El mismo ejemplo anterior, pero usando **sintaxis de consulta basada en métodos:**

Es importante destacar que en este tipo de sintaxis se usan las llamadas **expresiones lambdas**, observar: **(p => p.Cliente.Id == idCliente)**

```
private void ListarProductos(int idCliente)
{
    //1) Origen de Datos
    EFPW3 context = new EFPW3();

    //2) Consulta: Sintaxis de Metodo, con expresión lambda
    var productos = context.Producto.Where(p => p.Cliente.Id == idCliente).Select(p1 => p1);

    //3) Ejecución de Consulta
    foreach (Producto p in productos)
    {
        lblMensaje.Text += p.Nombre + " - ";
    }
}
```

Lazy Loading (Carga Perezosa)

Lazy loading es un patron de diseño comumente usado en programación a propósito de posponer la inicialización de un objeto hasta el momento en el cual este es necesitado. El objetivo es contribuir a la eficiencia; de esta manera los distintos objetos de una clase se irán cargando a medida que los vamos usando.

Estudiemos el siguiente ejemplo: si tenemos una entidad **Empleado,** que tiene como propiedad una entidad **DireccionEmpleado**, y hacemos la siguiente consulta LINQ to entities:

```
EFPW3 ctx = new EFPW3();

var emps = (from em in ctx.Empleados
            select em).ToList();

foreach (Empleado em in emps)
{
    string nombre = em.Nombre;
    string calle = em.DireccionEmpleado.Calle;
}
```

En la siguiente consulta no se recupera información para el objeto **DireccionEmpleado**

Al momento que se quiere acceder a la propiedad **DireccionEmpleado**, es justo en ese preciso instante cuando se recupera su información desde la BD.

```
public EFPW3() : base("name=EFPW3", "EFPW3")
{
    this.ContextOptions.LazyLoadingEnabled = true;
    OnContextCreated();
}
```

Constructor del Contexto:
Por defecto EF trabaja en esta modalidad **LazyLoading (ver)**

Eager Loading (Carga Temprana)

Eager Loading es lo opuesto a Lazy Loading, es decir los objetos relacionados se cargaran o recuperaran la primera vez, en el momento de ejecutar la consulta.

Para eager Loading debemos explicitar el metodo Include(), pasando como parámetro la-s entidad-es que queremos se recuperen con la consulta.

```
EFPW3 ctx = new EFPW3();

var emps = from em in ctx.Empleados.Include("DireccionEmpleado")
           select em;

foreach (Empleado em in emps)
{
    string nombre = em.Nombre;
    string calle = em.DireccionEmpleado.Calle;
}
```

Se explicita que se recuperará la entidad DireccionEmpleado en la consulta.

Podemos recuperar la consulta que EF ejecuta contra la base de datos con la siguiente línea de código:

```
((System.Data.Objects.ObjectQuery)emps).ToTraceString();
```

Ciclo de vida

En EF cada entidad tiene un estado (EntityState) que ira cambiando dependiendo de la operación que realice el contexto (DbContext).

El estado de la entidad es un tipo enum (System.Data.EntityState), que declara los siguientes valores:

•**Added:** Entidad agregada al contexto (Add).

•**Deleted:** Indica que la entidad esta eliminada del contexto (Delete)

•**Modified:** Cada vez que se modifica el valor de alguna propiedad de una entidad recuperada del contexto.

•**Unchanged**: Indica que la entidad original recuperada desde el contexto no ha sido modificada (escenario conectado)

•**Detached:** Indica que la entidad no esta ligada o no pertenece al contexto (Detach)

El contexto no solamente referencia a todos los objetos recuperados desde la base de datos, sino que mantiene ademas todo las modificaciones vinculadas a las propiedades de cada entidad. Esta caracteristica es conocida como **Change Tracking.**

Operaciones CRUD (SaveChanges)

Crear nuevos elementos (Create)

```
protected void Agregar()
{
        EFPW3 ctx = new EFPW3();

        //1. Nuevo Objeto

        Cliente cli = new Cliente();
        cli.Nombre = "Chevrolet S.A";
        ctx.Clientes.Add(cli);

        //2. Nuevo Objeto

        cli = new Cliente();
        cli.Nombre = "Radio Planeta";
        ctx.Clientes.Add(cli);

        ctx.SaveChanges(); //se persisten los datos en la BD, se
crearon dos clientes
  }
```

Actualizar elementos (Update)

```
protected void Actualizar()
 {
        using ( var ctx = new EFPW3())
        {

           Empleado emp = (from em in ctx.Empleados
                    select em).First();

           emp.Nombre = "Modificado";
    //Cuando se ejecuta la línea anterior, el entityState pasa a “Modified”

            ctx.SaveChanges();
        }
}
```

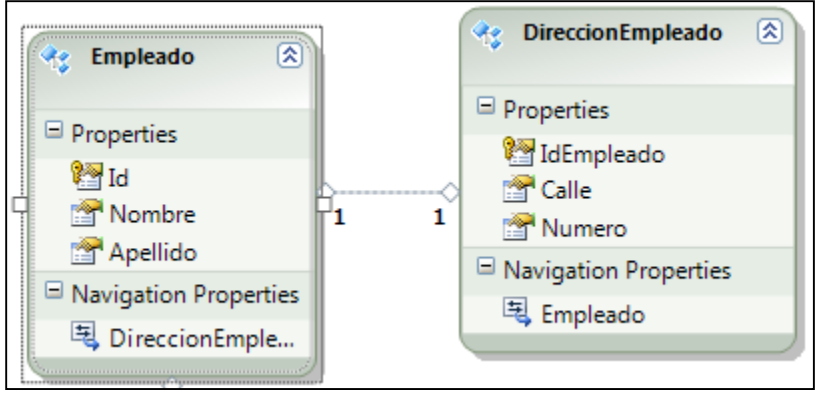
Eliminar elementos (Delete)

```
protected void Eliminar()
{
        EFPW3 ctx = new EFPW3();

        var clientes =context.Clientes;

        foreach (Cliente c in clientes)
        {
            ctx.Clientes.Remove(c);
        }
      ctx.SaveChanges();
}
```

Relaciones entre entidades (Uno a uno):

Cada entidad es una propiedad de la otra entidad. En el ejemplo siguiente se muestra que la entidad DireccionEmpleado es una propiedad de la entidad Empleado; al mismo tiempo la entidad Empleado es una propiedad de la entidad DireccionEmpleado:

Modelo Conceptual (1----1)



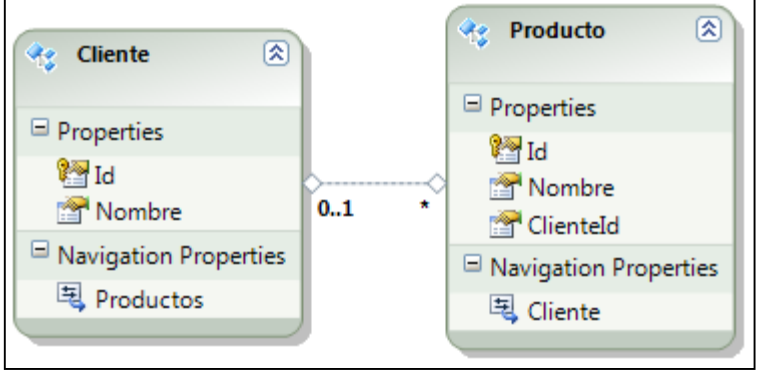
Ejemplo: Nuevo Empleado, nueva Dirección

```
protected void NuevoEmpleadoConDireccion()
 {
        EFPW3 ctx = new EFPW3();

        //Add, nuevo empleado nueva direccion.
        Empleado emp = new Empleado();
        emp.Nombre = "Joaquin J";

        DireccionEmpleado dir = new DireccionEmpleado();

        dir.Calle = "AV Independencia";
        dir.Numero = 3700;

        //
        emp.DireccionEmpleado = dir;

        ctx.Empleados.Add(emp);

        ctx.SaveChanges();
}
```

Relaciones entre entidades (Uno a Muchos):

En el modelo conceptual, en una relación uno a muchos, una entidad contiene una propiedad tipo colección o lista de elementos de otra entidad. En el ejemplo siguiente se muestra que la entidad Cliente contiene una propiedad lista de productos. A su vez, cada elemento Producto contiene una propiedad tipo entidad Cliente.

Modelo Conceptual (1----*)



Ejemplo: Nuevo Cliente, Nuevos Productos

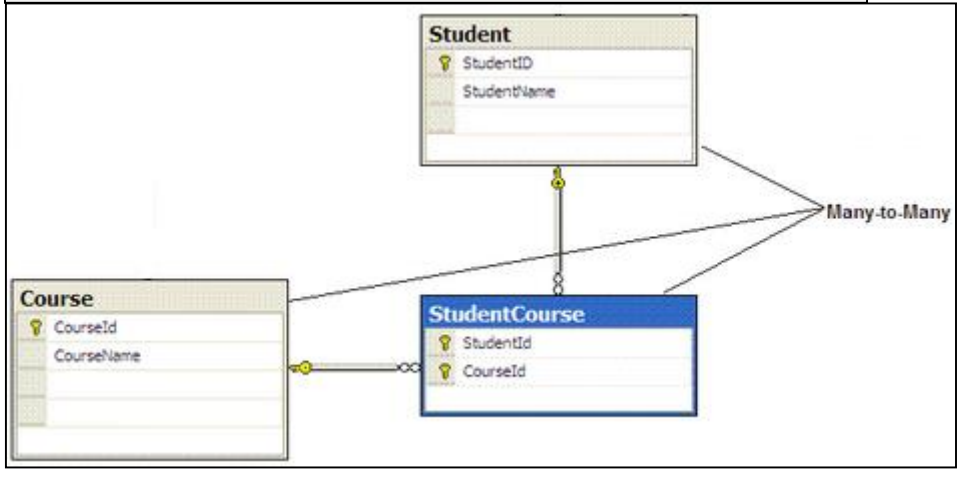
```
protected void NuevoClienteConProductos()
 {
        EFPW3 ctx = new EFPW3()

        //3. a) Entity relacion uno a muchos

        Cliente cli2 = new Cliente();
        cli2.Nombre = "Sancor SA";

        Producto pro = new Producto();
        pro.Nombre = "Leche Larga Vida";
        //pro.NombreFantasia = "Cualquier Cosa";
        cli2.Productos.Add(pro);

        //3. b) Relacion uno a muchos con partial
        cli2.Productos.Add(new Producto("Leche Descremada"));

        ctx.Clientes.Add(cli2);
        ctx.SaveChanges();
}
```

Relaciones entre entidades (Muchos a Muchos)

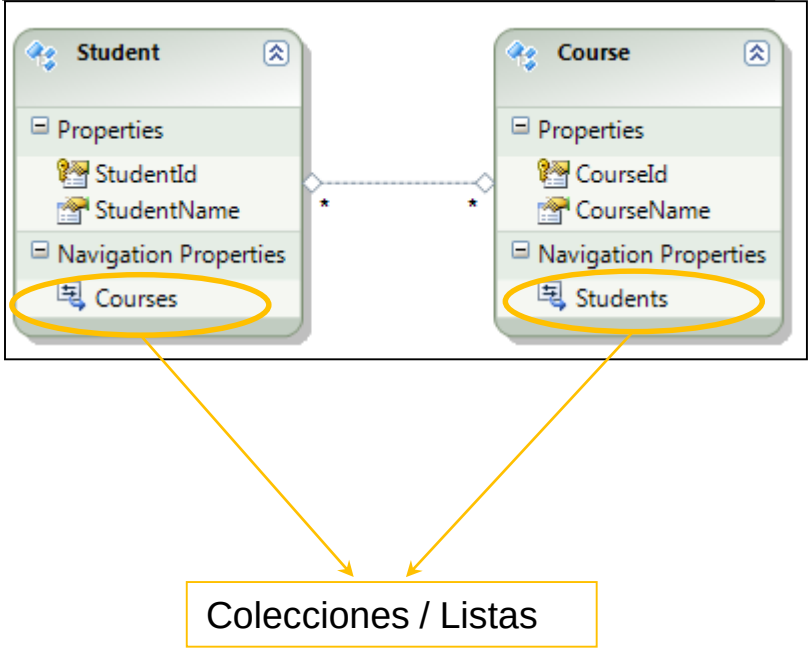
Como es de saberse, en el paradigma relacional, una relación muchos a muchos se representa agregando una tabla intermedia adicional entre dos tablas. De esta forma tenemos dos relaciones del tipo uno a muchos hacia esa nueva tabla adicional.

En el paradigma orientado a objetos, no es necesario agregar una entidad adicional, por el contrario, la relación entre ambas entidades se logra con dos colecciones o listas en cada entidad. De esta manera, en el ejemplo; la entidad Student tiene una colección de cursos y la entidad Cursos tiene una colección de estudiantes.

Modelo Relacional (muchos a muchos)



Modelo Conceptual (muchos a muchos)



Relaciones entre entidades (Muchos a Muchos)

En los siguientes ejemplos de código se muestran operaciones sobre este tipo de relación:

Se crea un nuevo grupo y se le asignan todos los empleados:

```
protected void NuevoGrupoConEmpleados()
{
   EFPW3 context = new EFPW3();

   Grupo g = new Grupo();
   g.Nombre = "Marketing Directo";

   var emps = context.Empleados;

   foreach (Empleado emp in emps)
   {
      g.Empleados.Add(emp);
   }
   context.Grupos.Add(g);

   context.SaveChanges();
}
```

Se eliminan todos los grupos de un Empleado:

```
protected void EliminarGrupos(int IdEmp)
{
   EFPW3 context = new EFPW3();

   var emp = (from e1 in context.Empleados.Include("Grupo")
              where e1.Id == IdEmp
              select e1).First();

  emp.Grupos.Clear();

   context.SaveChanges();
}
```

Se elimina un empleado de un Grupo:

```
protected void EliminarGrupo(int IdEmp, int IdGrupo)
{
   EFPW3 context = new EFPW3();

   var grupo = context.Grupos.Where (g1 => g1.Id == IdGrupo).FirstOrDefault();

   var emp = from e1 in context.Empleados
             where e1.Id == IdEmp
             select e1;

  grupo.Empleados.Remove(emp.First());
   context.SaveChanges();
}
```

Herencia

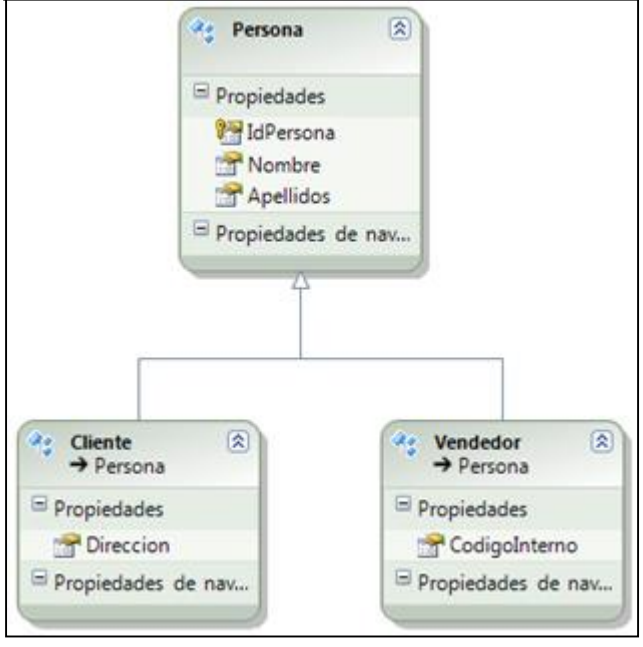
Existen dos soluciones convencionales para representar y mapear una relación de herencia entre los modelos relacional y POO (modelo conceptual). Estas soluciones son llamadas:

•Herencia por Tipo (**Table Per Type** -TPT)

•Herencia por Jerarquía (**Table Per Hierarchy** - TPH).

Cualquiera fuese la solución elegida, el modelo conceptual resultante será prácticamente el mismo, es decir lo que cambiara principalmente entre las dos soluciones es la representación relacional en la base de datos.

Modelo POO: Herencia

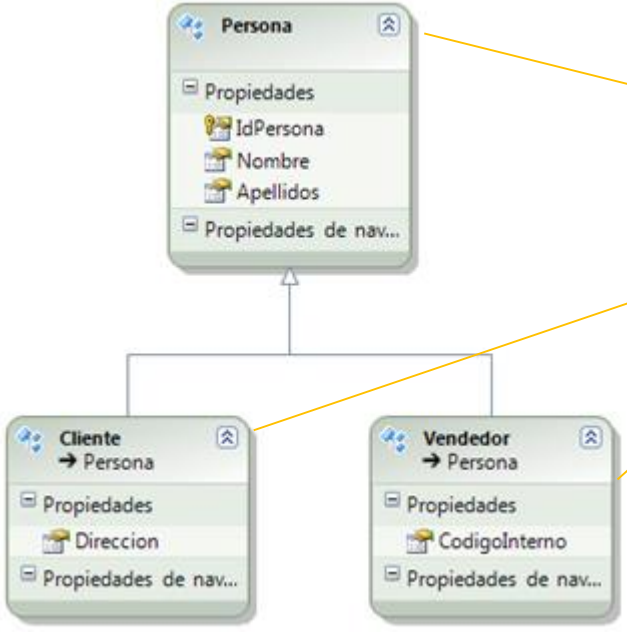


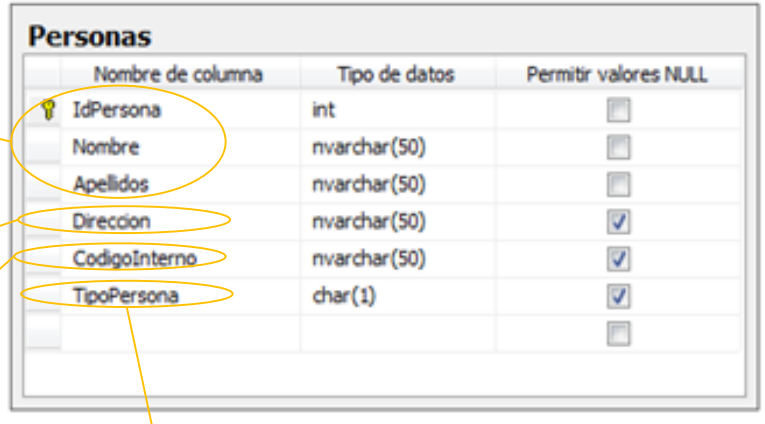
A continuación daremos una breve explicación de las dos soluciones

Herencia por Jerarquía (TPH)

En la solución TPH toda la jerarquía de herencia se guarda en una sola tabla, la cual tendrá campos correspondientes para las propiedades de la clase base y las propiedades de las clases derivadas. Además, es obligatorio incluir un campo discriminatorio para saber a qué tipo de entidad pertenece el registro activo.

Desventaja:
Como un registro guarda entidades de varios tipos, todos los campos que no pertenezcan al tipo de entidad activa tienen que ser nulos y siendo así, no podemos garantizar la coherencia del tipo de entidad activa puesto que no podemos, por ejemplo, jugar con la restricción de aceptar o no nulos en determinados campos de nuestra entidad puesto que todos los campos de la tabla deben de admitir nulos (excepto los campos de la entidad base y el campo discriminatorio).
Además, si hay muchos elementos en la jerarquía de herencia con muchas propiedades cada uno la tabla puede crecer desmesuradamente en tanto a su tamaño y gestión.





Campo discriminatorio: Agregar una condición desde el modelo conceptual. Ej:

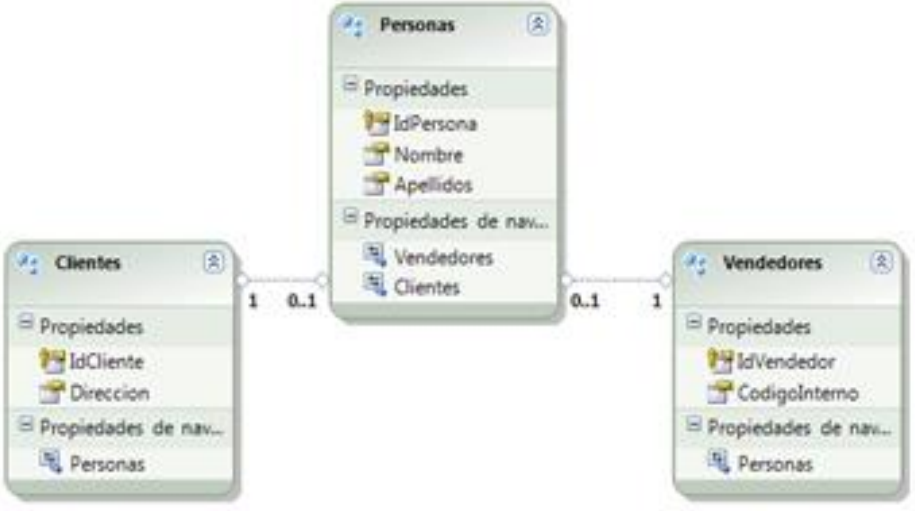
Persona – *When TipoPersona es NULL*
Cliente – *When TipoPersona es C*
Vendedor – *When TipoPersona es V*

Herencia por Tipo (TPT)

En la herencia por tipo, en el modelo relacional no tenemos una sola tabla sino que tenemos una tabla para los datos comunes (propiedades de la clase base) y una tabla por cada clase derivada en la jerarquía con sus propiedades concretas :

Para explicar los pasos para lograr el mapeo TPT seguimos un ejemplo:

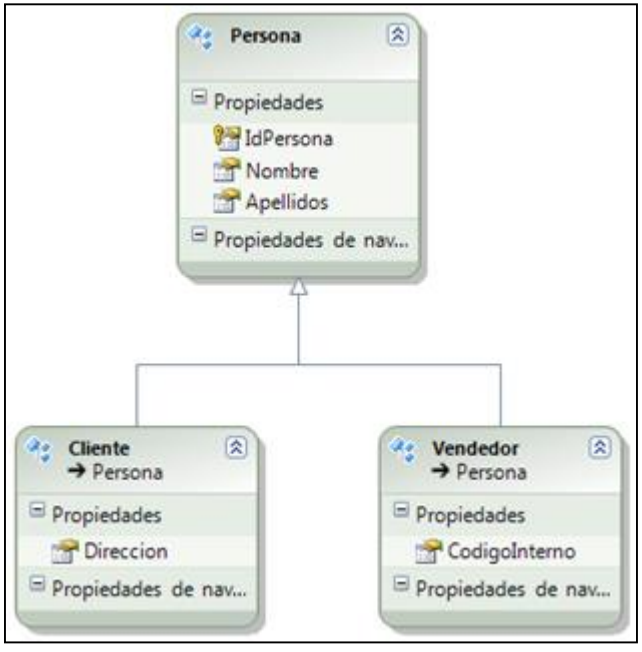
El modelo de dominio propuesto por Entity Framework si utilizamos Database First, será el siguiente:



Herencia por Tipo (TPT) (2)

1. **Eliminar** las asociaciones del modelo
2. **Crear** manualmente la relación de herencia entre clases derivadas y la clase base.
3. **Eliminar** las propiedades de clave de las clases derivadas. En nuestro caso, se eliminan IdCliente de la entidad Clientes e IdVendedor de la entidad Vendedores.
4. **Mapear** el campo IdCliente de la tabla Clientes al campo IdPersona de la tabla Personas.
5. **Mapear** el campo IdVendedor de la tabla Vendedores al campo IdPersona de la tabla Personas.

Finalmente, con cualquier solución (TPH o TPT) llegamos al mismo modelo de Entity Framework.



Licenciatura En Gestión Tecnológica

Programación Avanzada II

Formación Continua

Muchas gracias

Ing. Mariano Juiz