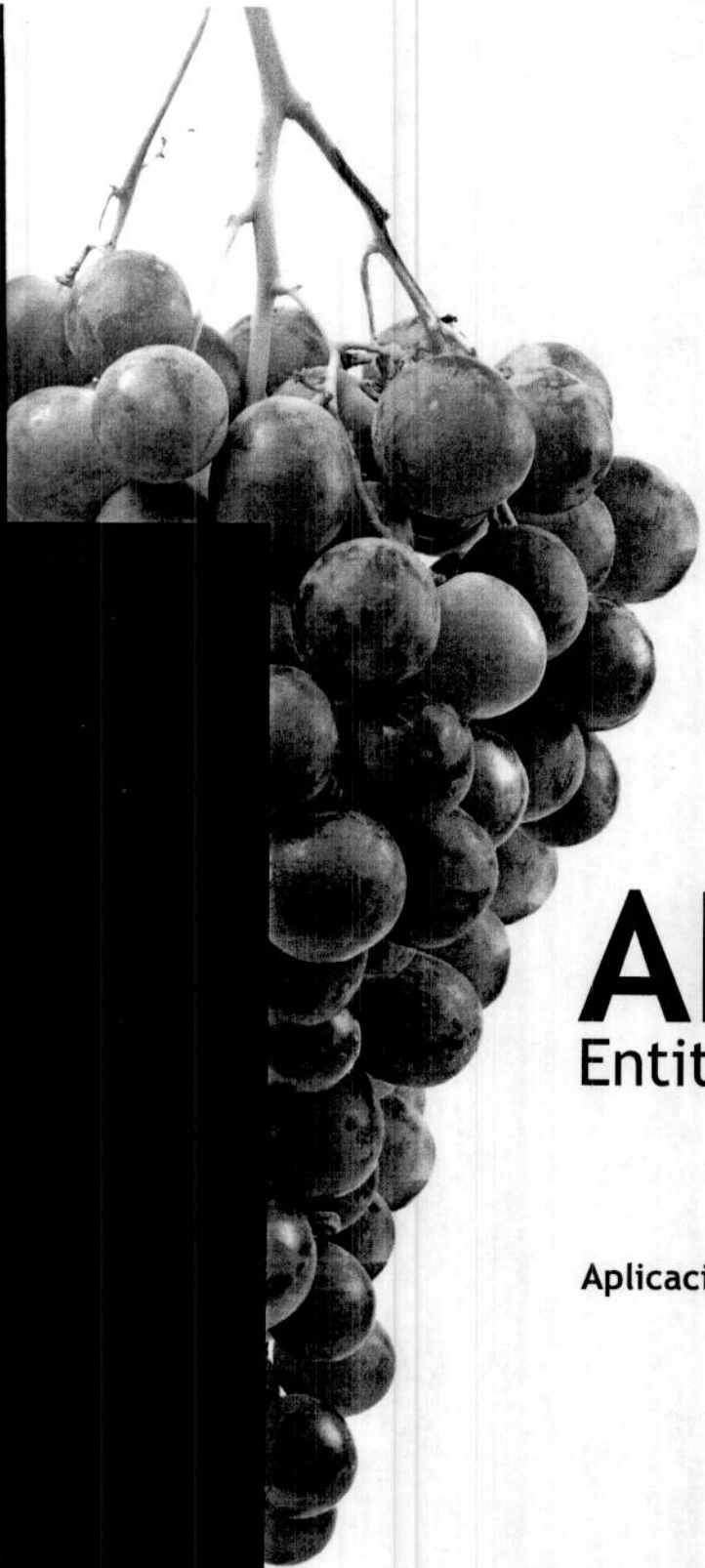


feed your brain®

ADO.NET Entity Framework 4.1

Aplicaciones y servicios centrados en datos

Unai Zorrilla Castro

Yamil Hernandez Saa
Cesar de la Torre Llorente
Pablo Peláez Aller

Prólogo de **José Manuel Alarcón, ASP.NET MVP.**

krasis PRESS

ADO.NET Entity Framework 4.1

Aplicaciones y servicios centrados en datos

Unai Zorrilla Castro

Colaboradores
Yamil Hernandez Saa
Cesar de la Torre Llorente
Pablo Peláez Aller



ADO.NET ENTITY FRAMEWORK 4.1 - APLICACIONES Y SERVICIOS CENTRADOS EN DATOS

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares del Copyright. Diríjase a CEDRO (Centro Español de Derechos Reprográficos, **www.cedro.org**) si necesita fotocopiar o escanear algún fragmento de esta obra.

DERECHOS RESERVADOS © 2011, respecto a la primera edición en español, por

Krasis Consulting, S. L.
www.Krasis.com

ISBN: 978-84-939036-3-3
Depósito Legal: C 1188-2011

Impreso en España-Printed in Spain

Agradecimientos

A mi mujer y a mi hija por su inmerecida paciencia con mi trabajo y mis viajes.

Unai

A mis padres, por su apoyo, protección, trabajo y sacrificios en todos estos años.

A mi hermano, por ser sin lugar a dudas el mejor del mundo.

A Marta, por todos los momentos juntos, por estar siempre a mi lado.

Yamil

Gracias a mi familia por todo el apoyo que me dan siempre

César

A mi mujer, artífice de mi mayor y mejor proyecto: mi hija.

Pablo

Contenido

AGRADECIMIENTOS iii

CONTENIDO iv

PRÓLOGO x

PRÓLOGO DE LOS AUTORES xiii

CAPÍTULO 1: INTRODUCCIÓN 1

1.- Introducción 1
1.1.- Modelos de dominio 1
1.1.1.- Las entidades en un modelo de dominio 2
1.1.2.- Las entidades no son todo en un modelo de dominio 4
1.1.3.- ¿Qué papel juega ADO.NET EF en todo esto? 4
1.2.- ¿Qué es ADO.NET Entity Framework? 5
1.2.1.- Arquitectura y componentes 6
1.2.2.- Proveedores específicos 7
1.2.3.- Entity Data Model 8
1.2.4.- Entity Client 19
1.2.5.- Object Services 21
1.2.6.- LINQ To Entities, L2E 21

CAPÍTULO 2: ENTITY DATA MODEL 23

1.- Introducción 23
1.1.- Elementos fundamentales de EDM 24
1.1.1.- Caso práctico: Las entidades 25
1.1.2.- Caso Práctico: Asociaciones 36
1.1.3.- Caso práctico: Table Splitting 49
1.1.4.- Caso práctico: Entity Splitting 54
1.1.5.- Caso práctico: La herencia 56
1.1.6.- Caso práctico: Defining Query 65
1.1.7.- Caso práctico: Entidades de solo lectura 68
1.1.8.- Caso práctico: Procedimientos almacenados 71
1.2.- Primero el modelo por favor 78
1.3.- Conceptos avanzados de EDM 86
1.3.1.- Model Defined Functions 86

1.3.2.- Tabla por tipo concreto y herencia 89
1.4.- Dividir lo grande 93
1.4.1.- Relaciones entre contextos 94

CAPÍTULO 3: ENTITY CLIENT 97

1.- Introducción 97
1.1.- Entity Client como proveedor de ADO.NET 98
1.1.1.- EntityConnectionStringBuilder 99
1.1.2.- Entity Connection 103
1.1.3.- EntityConnection y EntityDataReader 105
1.2.- Trabajando con Entity Client 108
1.2.1.- Jerarquía de tipos 108
1.2.2.- Consultas parametrizadas 115
1.2.3.- Consultas polimórficas 116
1.2.4.- Llamadas a procedimientos almacenados 118
1.2.5.- Revisión del código generado 119
1.2.6.- Transaccionabilidad 119
1.2.7.- Cache de planes de consulta 120

CAPÍTULO 4: OBJECTSERVICES Y LINQ TO ENTITIES 123

1.- Introducción 123
2.- Los servicios de objetos 123
2.1.- El contexto de trabajo 124
2.2.- Las entidades 130
2.2.1.- Selft Tracking Entities 135
2.2.2.- Entidades POCO 140
2.3.- ObjectSet<TEntity> 145
2.3.1.- Ejemplos de creación de consultas 148
2.3.2.- Métodos de construcción de consultas 152
2.3.3.- Navegación entre entidades 159
2.3.4.- Expansión de consultas 167
2.4.- LINQ To Entities 170
2.4.1.- Métodos de proyección y restricción 171
2.4.2.- Métodos de encuentro 175
2.4.3.- Métodos de partición 176
2.4.4.- Métodos de ordenación 177
2.4.5.- Métodos de agrupación 178
2.4.6.- Métodos de agregados 179
2.4.7.- Métodos de elementos y paginación 181
2.5.- Gestión del estado 183
2.5.1.- ObjectStateEntry 183
2.5.2.- Inserción de entidades 195
2.5.3.- Eliminación de entidades 199
2.5.4.- Actualización de entidades 208
2.5.5.- Transaccionabilidad en los servicios de objetos 210
2.6.- Selft Tracking Entities y la gestión del estado 211

1.2.- Las expresiones de consulta 357
1.3.- Reescritura de expresiones de consulta 361
1.4.- La (no) semántica de los operadores de consulta 363
1.5.- Resolución de llamadas a operadores 364
1.6.- Ejecución diferida 364
1.7.- Los operadores de consulta estándar 365
1.8.- El patrón de expresiones de consulta 366
1.9.- Sintaxis de las expresiones de consulta 367
1.10.- Tabla de operadores de consulta estándar 369
1.11.- Algunos ejemplos 372
1.12.- Extensiones de LINQ 379
1.13.- La interfaz IQueryable<T> 380
1.14.- ¿Qué hacen los operadores de IQueryable<T>? 381
1.15.- Sobre la disponibilidad de operadores y funciones 382
1.16.- Mecanismos de actualización 384

APÉNDICE B: REFERENCIA DE ENTITY SQL (ESQL) 385

1.- Introducción 385
1.1.- Diferencias con otros dialectos 385
1.2.- Sentencias de consulta 386
1.2.1.- FROM 386
1.2.2.- SELECT 387
1.2.3.- GROUP BY 390
1.2.4.- HAVING 390
1.2.5.- ORDER BY 391
1.2.6.- Expresiones 391

APÉNDICE C: ADO.NET ENTITY FRAMEWORK 4.1: ALGUNAS NOTAS DE RENDIMIENTO 409

1.- Introducción 409
2.- Consultas parametrizadas 409
3.- Precompilación de vistas 413
4.- LazyLoadingEnabled 415
5.- Capacidades de Modelado 417
6.- MergeOption 418
7.- Indices en las consultas 419
8.- Conclusión 422

APÉNDICE D: PLANTILLAS T4 423

1.- Introducción 423
2.- Las Plantillas t4 425
2.1.- Bloques de código 427
2.1.1.- Bloques de control de expresiones 427
2.1.2.- Bloques de control de características de clase 428
2.1.3.- Utilizar definiciones externas 429

2.6.1.- Entidades STE en Silverlight 220
2.7.- Entidades POCO y la gestión del estado 223
2.7.1.- POCO Proxies y Windows Communication Foundation 228

CAPÍTULO 5: EF 4.1 EN EL MUNDO REAL 233

1.- Introducción 233
2.- El dominio, las entidades y ef 4 234
2.1.1.- Definiendo las abstracciones 241
2.1.2.- Estableciendo la infraestructura 245
2.1.3.- "Testando" ADO.NET Entity Framework 4.1 253
2.2.- STE y las entidades duplicadas 259
3.- Conclusiones 266

CAPÍTULO 6: ENTITY FRAMEWORK 4.1 267

1.- Introducción 267
2.- Instalación y puesta en marcha 268
3.- Un nuevo modelo de trabajo 269
3.1.- Un primer ejemplo, los elementos fundamentales 269
3.2.- Las conexiones y el modelo de datos 271
3.2.1.- Modificación de los parámetros de conexión 273
3.2.2.- Extendiendo nuestros IDbConnectionFactory 274
3.2.3.- Inicializadores de conexión 275
3.3.- Convenciones y mapeos 277
3.4.- Convenciones por defecto 277
3.5.- Mapping 279
3.5.1.- Las entidades 280
3.5.2.- Tipos complejos 295
3.5.3.- Asociaciones 299
3.5.4.- Table Splitting 315
3.5.5.- Entity Splitting 318
3.5.6.- Herencia 319
3.5.7.- Elementos no soportados 325
3.5.8.- En la vida real... 326
4.- NUEVAS api 333
4.1.- DbContext nuestra nueva base 335
4.2.- DbEntityEntry 337
4.2.1.- Gestión del estado 338
4.2.2.- Gestión de las referencias 342
4.3.- DbSet - IDbSet 343
4.4.- Validación de entidades 347
5.- Conclusiones 353

APÉNDICE A: FUNDAMENTOS DE LINQ 355

1.- Introducción 355
1.1.- Presentación de LINQ 356

Prólogo

Seamos claros: la mayor parte los programadores odian las bases de datos. Si no las odian, al menos no les tienen, ni de lejos, el mismo "cariño" que le dispensan a sus algoritmos y otras partes del código. Hasta crear interfaces de usuario normalmente nos parece más grato que idear consultas para sacar información de un almacén de datos.

En gran parte esto se debe a que trabajar con bases de datos relacionales es completamente diferente a hacerlo con clases, interfaces y otras abstracciones de la programación orientada a objetos.

Muchos programadores se piensan que las bases de datos relacionales se llaman así porque existen una serie de tablas de datos (que podrían asimilarse a matrices) y éstas se "relacionan" entre sí, ya que se establecen relaciones de uno a muchos, muchos a muchos, etc.... En realidad el nombre se le otorga porque lo que generalmente llamamos "Tablas", en realidad, se denominan "Relaciones" en la notación del álgebra y cálculo relacionales que subyacen en toda la teoría de bases de datos, formulada por E.F. Codd en 1969. Así, cada conjunto de datos (tabla) es una relación, y una base de datos no es más que un conjunto de relaciones. Del mismo modo, lo que habitualmente llamamos los programadores "relaciones" son en realidad formas de vincular conceptualmente entre sí las relaciones de datos, por medio de campos comunes que permiten enlazar los elementos de una relación con los de otra. Finalmente, las consultas no son más que operaciones de conjuntos – matemáticas puras, al fin y al cabo- realizadas sobre los conjuntos de relaciones.

Todo ello tiene implicaciones importantes sobre el diseño de los almacenes de datos, que deben trabajar según la teoría de conjuntos. ¿Quién no ha oído hablar de que "hay que normalizar -o incluso desnormalizar. La base de datos?

Esto, que puede ser muy interesante, en realidad es ajeno a lo que solemos usar los programadores. Es mucho más natural para nosotros trabajar con clases e instancias de las mismas (objetos), y crear relaciones entre ellas bien mediante herencia, encapsulación o polimorfismo (los pilares de la POO), o simplemente haciendo que unos objetos referencien a otros a través de una propiedad. Así, es mucho más natural crear una clase "Factura" con una propiedad que sea una colección de objetos "LineasFactura", que tener las facturas en una matriz y las líneas de factura en otra y tener que preocuparnos de filtrarlas para obtener lo que necesitamos en cada caso.

Los intentos habituales de hacer encajar el modo tradicional de trabajo de la POO con el de los almacenes de datos subyacentes provocan multitud de problemas

2.1.4.- Utility Methods 430
2.1.5.- Directivas de Output 431
3.- Plantillas vs Plantillas Pre-Procesadas 431
3.1.- Plantillas pre-procesadas 432
4.- transformar datos y modelo 435

APÉNDICE E: EDMGEN 437

1.- Introducción 437

ÍNDICE ANALÍTICO 441

La primera edición de este libro fue el primer trabajo sobre el tema que se publicó en el mundo, en todos los idiomas. Salió a la venta el mismo día en el que Microsoft lanzó la tecnología. En esta segunda parte se ha rehecho por completo el contenido para integrar todos los cambios y novedades que incorpora ADO.NET Entity Framework 4.1.

¡Espero que lo disfrutes y sobre todo que aprendas!

José Manuel Alarcón
MVP de ASP.NET Development
Director de Krasis Press
@jm_alarcon

conocidos como "desajustes de impedancia". Al final lo que se ha hecho tradicionalmente ha sido trabajar con el lenguaje de consulta SQL -propio de las bases de datos relacionales- desde el código de nuestras aplicaciones, viéndonos forzados a usar conceptos muy alejados de lo que es natural en programación: trabajar con objetos.

¿No sería mucho más interesante pensar sólo en modelos de objetos sin preocuparnos del almacén de datos que hay debajo para sustentar su persistencia? Si queremos una factura creamos un objeto factura, que tiene la inteligencia suficiente para validar sus propios datos y su relación con las demás facturas, le añadimos objetos que representan líneas de facturas y que también tienen su propia inteligencia, y finalmente decimos "Guardar". Listo. Nos despreocupamos de cómo se almacena o se recupera toda esta información por debajo.

Este es el objetivo que persigue ADO.NET Entity Framework: abstraer al programador de conceptos que le son ajenos y que se pueda centrar en su trabajo, obviando las diferencias existentes entre la POO y el álgebra relacional.

Desde .NET 3.0, además, existe una sintaxis orientada a objetos para realizar consultas de manera integrada en el lenguaje: LINQ. Esta tecnología nos permite consultar datos independientemente de dónde estén ubicados para realizar exactamente del mismo modo una consulta contra una colección de objetos en memoria, un archivo XML, una base de datos o, incluso, un servicio on-line o un repositorio de código.

Si sumamos la potencia de LINQ a la abstracción sobre las bases de datos relacionales que nos proporciona ADO.NET Entity Framework 4.1 podremos obtener grandes ventajas a la hora de desarrollar aplicaciones: rapidez, separación de lógica y almacenamiento, independencia de la base de datos, "testabilidad", facilidad de mantenimiento...

Pero en esta vida todo tiene un precio. Y en este caso el peaje a pagar es el aprendizaje que requiere dominar la tecnología y conocer cómo funciona "bajo el capó" para sacarle todo el partido. Y eso es precisamente lo que nos enseñan los autores en su libro.

No se trata de un libro de "recetillas" para salir del paso, sino que profundiza en el funcionamiento de la tecnología para explicarnos el porqué de cada cosa. En este sentido si no tienes experiencia previa seria como programador de datos puede que al principio te sientas un poco perdido, pero al final te acabarás ubicando con EF.

Así, podrás utilizarlo para aprender los rudimentos del uso de EF, pero querrás tenerlo a mano para cuando haya que ir más allá y haya que salvar limitaciones. Está salpicado de consejos prácticos sacados de las mejores prácticas sobre arquitectura de aplicaciones empresariales, que salen de la experiencia de sus autores en el desarrollo de proyectos reales para todo tipo de empresas, incluyendo las más importantes multinacionales.

Y qué decir de los autores... Son los principales expertos del sector en esta tecnología. No te puedes hacer una idea del esfuerzo que me ha llevado como editor lograr que se comprometieran a crear esta obra. Pero el resultado seguro que ha merecido la pena :-)

Prólogo de los autores

A sabiendas de que es una frase muy socorrida, la escritura de un libro siempre es una tarea ardua y complicada. Si tenemos en cuenta además que esta es una segunda versión, ya sabe lo que se dice de las segundas versiones, entonces, se imaginará la responsabilidad que hemos tenido a la hora de decidir la escritura del mismo.

Sinceramente esperamos no defraudar al lector, tanto al que no ha tenido la oportunidad de leer la primera versión -y por lo tanto esperamos que el contenido le sea suficiente para afrontar un proyecto con esta tecnología-, como aquel que ha leído la versión anterior y en esta busque los nuevos elementos incluidos en ADO.NET Entity Framework 4.1.

Contenido del libro

El libro consta de cinco capítulos más cuatro apéndices de referencia:

- Los primeros capítulos cubren ampliamente la mayoría de las características de ADO.NET Entity Framework 4.1. Desde los elementos básicos referentes a las herramientas de Visual Studio 2010, hasta el uso de ADO.NET Entity Framework en el mundo real.

- Los apéndices de este libro permitirán profundizar al lector en diferentes elementos como la referencia de Entity SQL o el apéndice referido a T4.

Sobre los ejemplos

El código fuente de todos los ejemplos del libro, así como los fragmentos SQL, se encuentran disponibles para su descarga en el sitio Web del editor, **www.krasispress.com**. Para compilarlo y ejecutarlo, deberá tener instalada cualquier edición de Visual Studio 2010 o incluso Visual C# Express 2010.

CAPÍTULO

Introducción

1.- INTRODUCCIÓN

Para ser sincero con usted amigo lector, la escritura de este libro no ha resultado nada fácil. Después del primer libro, dedicado a **ADO.NET Entity Framework 1.0**, escribir sobre las novedades que la segunda versión incluye, no solamente implica un reto al explicar nuevos y variados conceptos sino también el trabajo de revisar aquellos elementos ya comentados en la edición anterior de esta obra. Al final, aunque la estructura es similar al libro anterior, el contenido sufre un cambio importante, ampliando aspectos no profundizados anteriormente, y por supuesto incluyendo las distintas novedades que la nueva versión nos ofrece.

Este primer capítulo está divido en dos partes fundamentales. En la primera introduciremos al lector en el concepto de "modelos de dominio". Aunque se ha extendido y profundizado en el tema con respecto al libro anterior, esta introducción no es más que una peq-ueña reseña de ciertos conceptos y como aplican los mismos en **ADO.NET Entity Framework**, pero no es para nada una introducción a DDD (*Domain Driven Design*) ni hacia la "práctica" de DDD. La segunda parte de este capítulo es una introducción a la arquitectura de EF y los distintos módulos funcionales que la componen.

1.1.- Modelos de dominio

Desde las primeras charlas y eventos en los que tuve la oportunidad de presentar **ADO.NET Entity Framework** siempre rondó por mi cabeza la necesidad de introducir a los asistentes en los modelos de dominio y en la impedancia que los lenguajes orientados a objetos tienen con los modelos relacionales y viceversa. Este trabajo, que en las alturas tecnológicas que estamos parecería innecesario, es indispensable para muchos desarrolladores de aplicaciones.

Nota: Aunque la definición de modelo tiene muchas acepciones, en nuestro ámbito un modelo es cualquier abstracción del mundo real que puede tener diferentes presentaciones como diagramas, clases etc…

El porqué de clarificar qué es un "modelo de dominio" y como crear un dominio surge realmente de la necesidad de separar los conceptos de negocio, información sobre la situación de los mismos, así como de las reglas que los guían de otros aspectos de tipo infraestructura o implementación, que en realidad nada tienen que ver con ellos. Si usted es un programador tradicional de ADO.NET seguramente, podrá observar como en sus desarrollos el modelo relacional prima o guía los trabajos que tiene que hacer, y limita las capacidades de la orientación a objetos. El motivo es que algunos elementos como la separación de responsabilidades, la ignorancia de la persistencia y otros conceptos son, sino imposibles, sí muy difíciles de realizar.

Aunque no es objeto de este libro, a lo largo del mismo se pondrán de manifiesto distintos patrones habituales en arquitecturas orientadas al dominio, así como técnicas para mantener de forma correcta los principios de desarrollo más habituales en este tipo de soluciones. No obstante, me gustaría clarificar que aquí nada se hablará con respecto a DDD o Domain Driven Design, ni como 'practicarlo' en un desarrollo. Es decir, se establecerán y desarrollarán patrones habituales en DDD pero no se hablará sobre aspectos teóricos ni organizativos o artefactos, como el lenguaje ubicuo etc…

1.1.1.- Las entidades en un modelo de dominio

Las entidades representan las abstracciones dentro de un modelo de dominio. Normalmente, tienen una correspondencia directa con los objetos principales dentro de un negocio, como por ejemplo podría ser un cliente, un pedido etc.

Uno de los elementos principales de los que tiene que disponer una entidad es el de la 'identidad y de la continuidad de la misma'. De forma general con identidad nos referimos al elemento que permite identificar a las entidades de forma unívoca dentro de un conjunto de las mismas. De forma general se tiende a asociar la identidad con los valores de atributos o propiedades, aunque esto no tenga que ser así puesto que podría darse el caso de que dos entidades tengan los mismos valores de atributos pero identidades diferentes. Muchos elementos, en el dominio real (la realidad del negocio) o en el modelo de dominio de la aplicación (abstracción del negocio), están definidos por su identidad y no por sus atributos.

Un muy buen ejemplo de entidad es una persona. Los atributos de las personas pueden ir cambiando a lo largo de su vida, como la dirección, datos financieros e incluso el nombre, y sin embargo, continúa siendo la misma entidad, la misma persona, en este ejemplo. Por lo tanto, el concepto fundamental de una ENTIDAD es una vida continua abstracta que puede evolucionar a diferentes estados y formas, pero que siempre será la misma entidad.

Como habrá podido comprobar, el concepto de entidad resulta de trascendental importancia dentro de los modelos de dominio. El diseño de los mismos y las capacidades de cada uno para diferenciar qué es una entidad y que no lo es (por regla general definido como **Value Object**, elemento que contiene atributos pero no identidad) es vital para obtener buenos resultados.

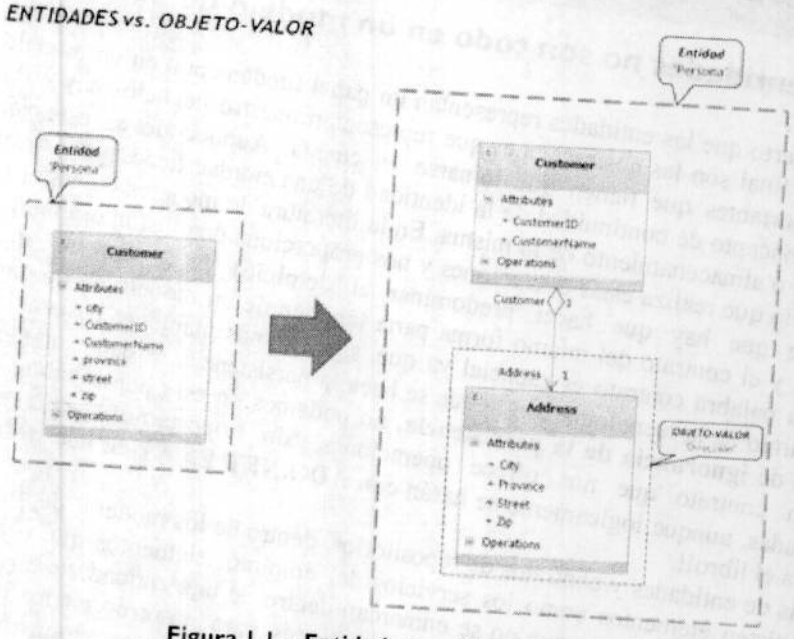


Figura 1.1.- Entidades vs. Objeto-Valor

Cuando se trabaja con un **ORM** (*Object Relational Mapping*) -como por ejemplo ADO.NET Entity Framework 4.1- uno de los elementos que forma parte de nuestro trabajo es precisamente el de extraer el modelo de entidades a partir de una base de datos existente, o bien el de crear directamente un modelo de entidades sin importarnos si existe o no estructura relacional subyacente. Lógicamente al final lo que tendremos será algo que tenga sentido en un lenguaje orientado a objetos, una estructura, una clase, una clase que implementa una interfaz o hereda de alguna clase base etc.

El cómo estas entidades son generadas es algo importante y da lugar a nuevos pensamientos y formas de abordar una solución. Piense por ejemplo el impacto que tiene para el diseño de una solución una entidad que hereda de una determinada clase, la cual está restringiendo la tecnología con la que se puede persistir frente al hecho de que podamos disponer de entidades que no necesiten, por aspectos de infraestructura, implementar o heredar de ninguna clase base. A lo largo de los siguientes capítulos iremos viendo las distintas opciones que ADO.NET Entity Framework 4.1 pone a nuestra disposición con respecto a la generación / creación de entidades.

Nota: Tal y como se ha comentado en un párrafo anterior, este libro no tratará sobre DDD y ni siquiera tiene pensado ser una guía de arquitectura. No obstante se hacen determinadas introducciones a ciertos conceptos importantes. Si quiere saber más y profundizar le recomiendo las siguientes lecturas:

. "Domain Driven Design" por Eric Evans
. "Patterns Of Enterprise Application Architecture" por Martin Fowler

1.1.2.- Las entidades no son todo en un modelo de dominio

Si bien es cierto que las entidades representan un papel fundamental en un modelo de dominio -al final son las abstracciones que representan nuestro negocio- hay otros elementos importantes que tienen que tomarse en cuenta. Aunque no es esencial, realmente el concepto de continuidad de la identidad de una entidad tiene que ver con la persistencia o almacenamiento de la misma. En la literatura de modelos de dominio, al intermediario que realiza estas operaciones y nos proporciona una visión orientada a objetos (note que hay que hacer predominar el dominio) se le conoce como Repositorio[1], y el contrato del mismo forma parte también de un modelo de dominio. Fíjese que la palabra contrato es esencial ya que, si queremos mantener desacoplado nuestro dominio de la tecnología con la que se hace la persistencia, es decir, mantener el principio de **ignorancia de la persistencia,** no podemos en este punto hablar más que de un contrato que nos ofrece operaciones, sin importarnos como estén implementadas, aunque lógicamente se harán con **ADO.NET EF 4.1** en nuestro caso, ¡¡de esto va el libro!!.

Además de entidades y contratos de repositorios, dentro de los modelos de dominio también existen elementos como los servicios del dominio, elementos que permiten crear y exponer operaciones que no se enmarcan dentro de una entidad en particular. Lógicamente estos elementos no son tan importantes para nosotros en lo que nos ocupa, Entity Framework, por lo tanto sugerimos al lector más interesado profundizar sobre estos elementos en las distintas referencias bibliográficas recomendadas.

1.1.3.- ¿Qué papel juega ADO.NET EF en todo esto?

Puede que llegado hasta aquí y durante el proceso de lectura haya vuelto hacia la carátula de este libro y revisara que es lo que estaba leyendo, puesto que hasta ahora apenas se ha hablado nada de Entity Framework. No se extrañe ni se enfade, la introducción anterior nos permite enseñar algunos conceptos importantes de los que se hablará en capítulos posteriores, como las plantillas de generación de entidades con

[1] Este patrón está definido en el libro Patterns Of Enterprise Application Architecture de Martin Fowler.

clases prescriptivas, objetos POCO o las técnicas para crear y testear Repositorios con ADO.NET EF.

1.2.- ¿Qué es ADO.NET Entity Framework?

ADO.NET Entity Framework es un marco de trabajo para la plataforma .NET que nos permite superponer varias capas de abstracción sobre un almacén relacional con el fin de hacer posible una programación más conceptual (basada en los conceptos del dominio con el que se trabaja) y de reducir a una mínima expresión el desajuste de impedancias causado por las diferencias entre los modelos de programación relacional y orientado a objetos.

Por una parte, es de destacar que EF es una parte integral de ADO.NET a partir de .NET Framework 3.5 SP1, lanzamiento de la primera versión, y que en Visual Studio 2010 ya disponemos de la versión 4.1. De todos modos este indicador de versión es 'marquetiniano' ya que realmente estamos en la segunda versión del producto.

Más exactamente, EF incluye un nuevo proveedor de ADO.NET, llamado Entity Client, que habilita el acceso a los modelos conceptuales. A lo largo de este libro, iremos viendo la uniformidad conceptual entre este proveedor y el resto de proveedores de ADO.NET.

Por otra parte, como todo marco de trabajo, EF incluye dos componentes fundamentales:

I. **Recursos para el entorno de desarrollo** y en particular un asistente para el diseño visual de modelos de entidades dentro de Visual Studio así como la generación de código a partir de los mismos.

II. **Biblioteca.** Los tipos que componen ADO.NET EF se implementan físicamente en el ensamblado *System.Data.Entity*. La organización lógica de eso tipos es tal y como se muestra en la tabla a continuación. Todo esto refuerza la idea de la pertenencia de EF a la familia de ADO.NET

Tabla 1.- Espacios de nombres relacionados con EF.

Concepto 1	Concepto 2
Espacio de nombres	Contenido añadido por EF
System.Data	Tipos básicos de Entity framework como EntityState, EntityObject…
System.Data.Common	Tipos relacionados con la implementación de proveedores de EF
System.Data.Common. CommandTrees	Tipos que se utilizan en los árboles de expression de LINQ to Entities
System.Data.Mapping	Tipos que se utilizan en el mapeo entre entidades lógicas y físicas.
System.Data.Metadata.Edm	Tipos permiten acceder a los metadatos

	de los modelos.
System.Data.Objects	Tipos que implementan el modelo de programación que hace posible un trabajo basado en objetos contra datos provenientes de una base de datos relacional. En particular, en este espacio de nombres reside la clase ObjectQuery<T>, que implementa IQueryable<T> y por lo tanto sirve como origen para las consultas integradas sobre modelos de entidades.
System.Data.Objects.Classes	Tipos que representan los principales conceptos que se utilizan en los modelos. Normalmente, no es necesario interactuar directamente con las clases de este espacio, pues el generador de código crea versiones más específicas de ellas

1.2.1.- Arquitectura y componentes

ADO.NET Entity framework 4.1 se apoya en seis elementos fundamentales construidos encima de toda las basesde ADO.NET, tal y como podemos ver en la siguiente figura.

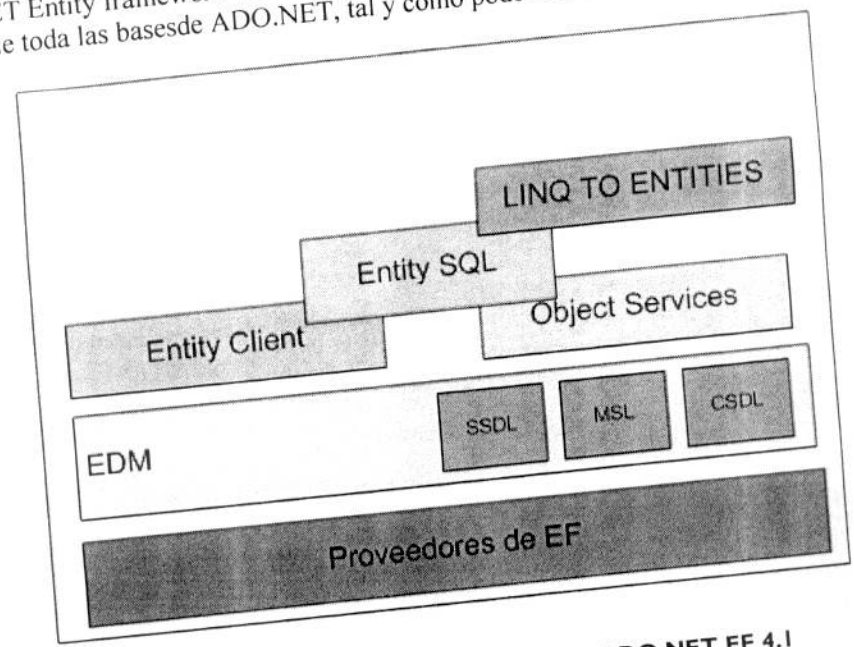


Figura 1.2.- Componentes principales de ADO.NET EF 4.1

A continuación, presentamos los conceptos fundamentales relacionados con cada uno de estos elementos (de abajo hacia arriba), profundizando en cada uno de ellos en los posteriores capítulos que acompañan este libro.

1.2.2.- Proveedores específicos

Una de las características más atractivas de ADO.NET EF es su "agnosticismo" con relación a la base de datos contra la que se trabaja. En verdad, EF es una implementación de un Data Mapper[2] entre las entidades definidas en un modelo conceptual y el esquema físico de la base de datos subyacente. Más adelante, veremos cómo el trabajo que se realiza sobre estas entidades es traducido directamente al dialecto específico de la base de datos con la que estemos trabajando gracias a los **proveedores de datos de ADO.NET Entity Framework.** Por defecto, en la actual versión de esta tecnología tendremos un proveedor específico para Sql Server, en todas sus versiones 2000, 2005 y por supuesto 2008 y 2008 R2, y también para SQL CE Compact Edition, la base de datos *InProc* que Microsoft pone a nuestra disposición[3].

Nota: Si no conoce Sql Server Compact Edition o bien tiene interés en conocer más acerca de qué es y cómo funciona esta base de datos 'in process', le recomendamos la lectura del libro de nuestro colega y amigo **José M. Torres** titulado "SQL Server 2008 Compact Edition. Aplicaciones Smart-client para escritorio y dispositivos móviles", publicado por la misma editorial de este libro y disponible en **www.krasispress.com**.

La posibilidad de crear aplicaciones multi-base de datos es cada vez más un requisito indispensable para fabricantes de software, también conocidos por sus siglas ISV (*Independent Software Vendors*), y departamentos de informática de grandes corporaciones dónde la selección del motor de la base de datos podría variar entre cambios organizativos y de directores. Lógicamente, el hecho de que las aplicaciones puedan soportar múltiples motores relacionales es un valor añadido para aquellas empresas que quieran 'colocar' su producto en el mercado. La posibilidad de implantar su producto en una empresa que usen como norma SQL Server pero también poder implementar el mismo en otra empresa con Oracle es algo que les puede proporcionar pingues beneficios.

Ya desde la primera versión del producto empezaron a desarrollarse distintos proveedores de Entity Framework, bien por parte de *partners* de Microsoft o por la comunidad, y distintos motores de bases de datos disponen de proveedores con los que poder trabajar. En los momentos de escribirse este libro la lista aproximada de proveedores disponibles puede verse en la siguiente tabla:

---

[2] Data Mapper en Patterns of Enterprise Application Architecture, Martin Fowler

[3] Solamente está soportada Sql CE Compact Edition 3.5 SP1 o superior en versión escritorio, **no móvil**.

Tabla 2.- Lista de algunos proveedores de ADO.NET EF 4.1

Base de datos	Proveedor
Oracle	○ DevArt DotConnect for Oracle ○ DataDirect Oracle Provider ○ ODP.NET Oracle, en fase Beta
MySQL	○ MySQL Connector ○ DevArt DotConnect for MySQL
SQLLite	○ DevARt DotConnect for SQLLite ○ System.Data.SqlLite
PostgreSQL	○ DevArt DotConnect for PostgreSQL

Lógicamente, la creación de un proveedor específico es algo que se escapa al propósito de este libro, pero si tiene interés en entender el proceso de implementación de un proveedor, puede revisar el proyecto **EFSampleProvider** (http://code.msdn.microsoft.com/EFSampleProvider), donde podrá encontrar una implementación de ejemplo, un foro de discusión y abundante documentación sobre el tema.

1.2.3.- Entity Data Model

El primero de los elementos fundamentales en la arquitectura de ADO.NET Entity Framework es el Modelo de Entidades, normalmente conocido como EDM. Este 'diseñador' nos permite definir los conjuntos de entidades y relaciones entre las mismas de nuestros modelos conceptuales, así como especificar de qué manera estos tipos se mapearán a la estructura de la fuente de almacenamiento relacional subyacente.

Para apoyar a EDM, disponemos de una serie de herramientas integradas dentro del entorno que nos ayudarán a crear nuestros modelos conceptuales. A continuación pasaremos a detallar el funcionamiento y objetivo de cada una de ellas:

1. Diseñador de modelos EDM (Entity Data Model Designer)

El diseñador de modelos es una herramienta visual integrada dentro de Visual Studio 2010 que permite crear y editar modelos conceptuales. Este diseñador está formado por los siguientes componentes:

- Una superficie de diseño para crear y editar los modelos de una forma rápida similar al trabajo con los diagramas de clases de Visual Studio.
- Una ventana de 'Detalles de mapeo', que nos permitirá ver y editar los mapeos entre los elementos del modelo conceptual y del esquema de la base de datos con la que estemos trabajando.
- Una ventana de 'Navegación por el modelo', en la cual podremos ver árboles de información sobre el modelo conceptual y el modelo físico.
- Nuevos elementos dentro de la ventana de herramientas, que nos permitirán, por ejemplo, crear las entidades, las asociaciones o relaciones de herencia.

El diseñador de modelos opera sobre ficheros de tipo **edmx**. Estos ficheros se forman mediante la combinación de tres secciones de metadatos en formato XML (que en ocasiones pueden presentarse también como ficheros independientes), llamadas respectivamente SSDL, CSDL y MSL.

- SSDL (*Storage Schema Definition Language*) describe la estructura física de la base de datos subyacente, incluyendo la definición de las tablas, columnas, vistas, procedimientos almacenados y relaciones entre los distintos objetos de la misma.
- CSDK (*Conceptual Schema Definition Language*) describe las entidades que deseamos tener en nuestro modelo conceptual, así como las propiedades de navegación o asociaciones entre las distintas entidades.
- Para terminar, el archivo edmx contiene la sección MSL (*Mapping Schema Language*), también conocida como sección C-S, mediante la cual se especifican cómo se asocian o relacionan las entidades del modelo conceptual (definido en la sección CSDK) con las tablas, vistas, columnas, etc. Del modelo físico definido en la sección SSDL.

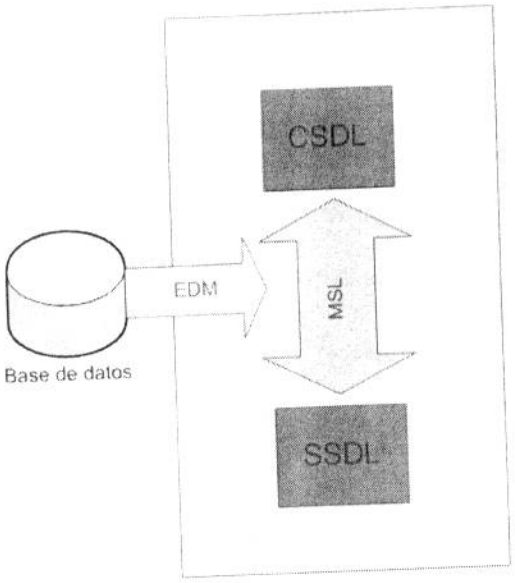


Figura 1.3.- Secciones de metadatos de un modelo EDMX

Una vez que se agrega dentro de un proyecto de Visual Studio 2010 el elemento ADO.NET Entity Data Model, el asistente de modelos (del que hablaremos a continuación) crea un nuevo archivo **edmx** y lo agrega al proyecto en el que estemos trabajando. La siguiente figura, nos muestra un ejemplo de un diseñador EDM y las distintas ventanas de herramientas con las que podemos trabajar.

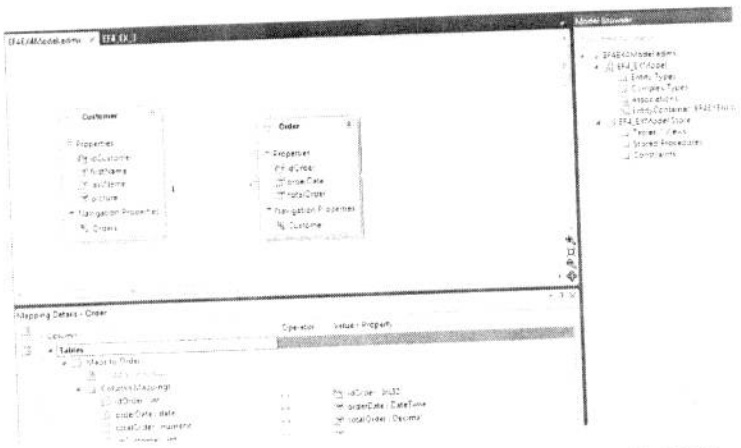


Figura 1.4.- Diseñador de EDM integrado en Visual Studio 2010

2. Asistente de modelos de entidades (Entity Data Model Wizard)

Tal y como hemos mencionado anteriormente, el asistente de modelos es el encargado de generar el archivo **edmx** con el que trabaja el diseñador. Para ello, este asistente permite crear el modelo a partir de una base de datos ya existente (la práctica más habitual aunque no la única como veremos a lo largo del siguiente capítulo, en la última parte).

Con el fin de mostrar el asistente mencionado realizaremos un sencillo ejemplo, en realidad, el propósito del mismo es más enseñarle y que conozca las herramientas que hablar de modelos de entidades o las distintas posibilidades que EDM nos ofrece, puesto que, esto, se tocará en el siguiente capítulo con una dedicación exclusiva. Para nuestro ejemplo, partiremos, de una base de datos con el esquema relacional que se puede ver en la figura 1.5.

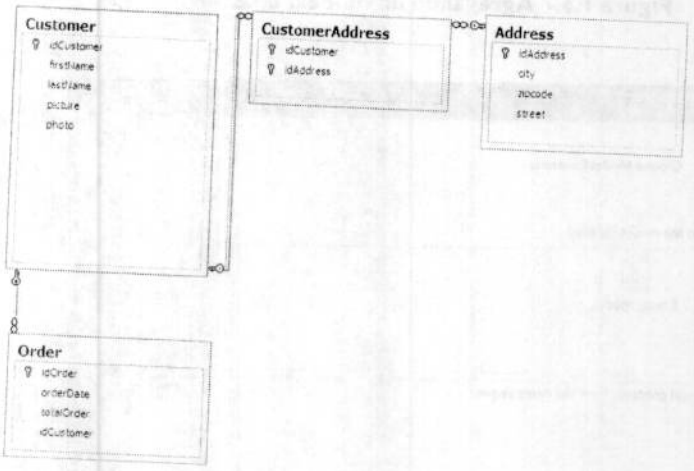


Figura 1.5.- Diagrama relacional del ejemplo

En un proyecto cualquiera de Visual Studio (un ejemplo de aplicación de consola le podrá valer para descubrir las herramientas con las que trabajaremos), agregamos un nuevo elemento de tipo ADO.NET Entity Data Model, Figura 1.6. Una vez hecho esto, el asistente nos permitirá seleccionar la base de datos a partir de la cual crearemos nuestro modelo conceptual, figuras 1.7 y 1.8.



Figura 1.6.- Agregando un modelo de EDM

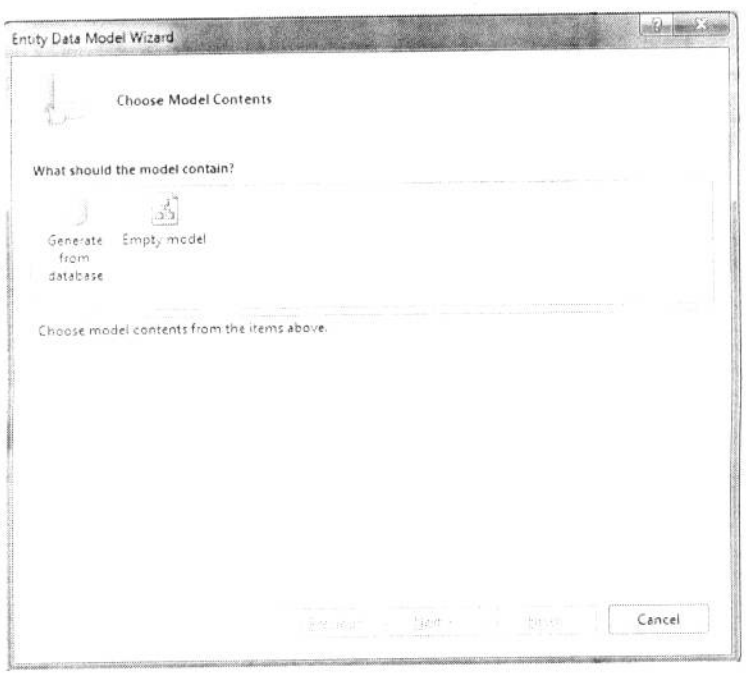


Figura 1.7.- Asistente de creación del modelo conceptual

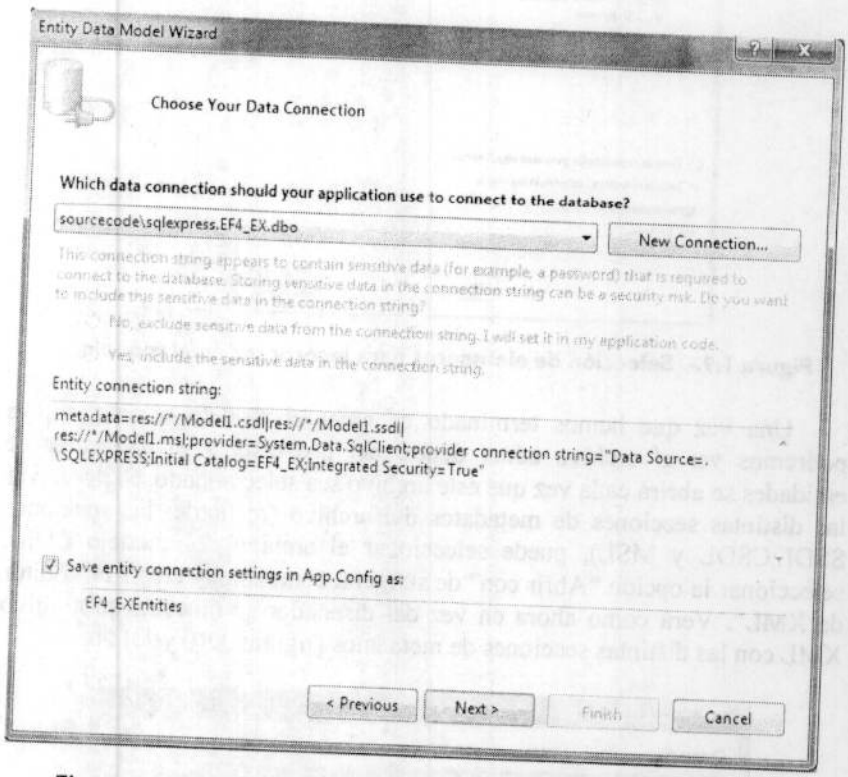


Figura 1.8.- Asistente de creación del modelo conceptual (2)

Nota: Aún a riesgo de parecer pesado, puesto que ya se ha comentado, me permito recordarle que la opción de crear un modelo de entidades directamente, sin partir de un esquema relacional, será tratado en todo detalle en la última sección del siguiente capítulo.

Para terminar, seleccionaremos los elementos de la base de datos que queramos que formen parte de nuestro modelo conceptual. En nuestro caso, simplemente ponemos las tablas de *Customer* y *Order*, Figura 1.9

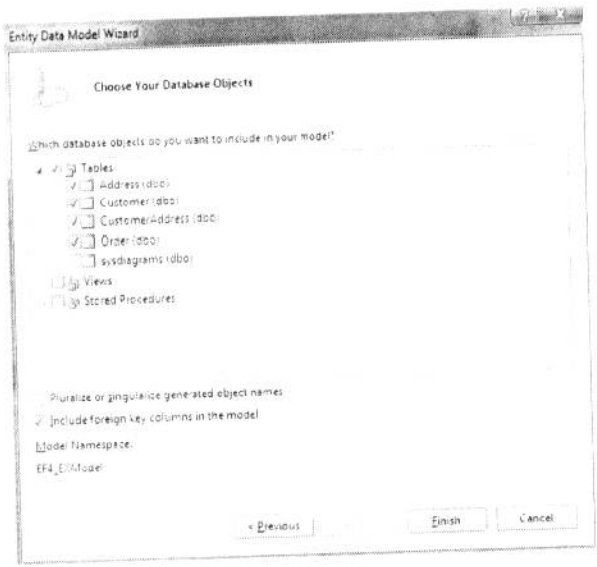


Figura 1.9.- Selección de elementos para incorporar en el modelo

Una vez que hemos terminado de crear el modelo conceptual, ya podremos ver el archivo edmx dentro del proyecto, y el diseñador de entidades se abrirá cada vez que este archivo sea seleccionado. Si desea ver las distintas secciones de metadatos del archivo (recuerde: las secciones SSDL,CSDL y MSL), puede seleccionar el archivo del modelo EDM, seleccionar la opción "Abrir con" de su menú contextual y elegir el "Editor de XML". Verá como ahora en vez del diseñador se muestra un archivo XML con las distintas secciones de metadatos (figuras 1.10 y 1.11).

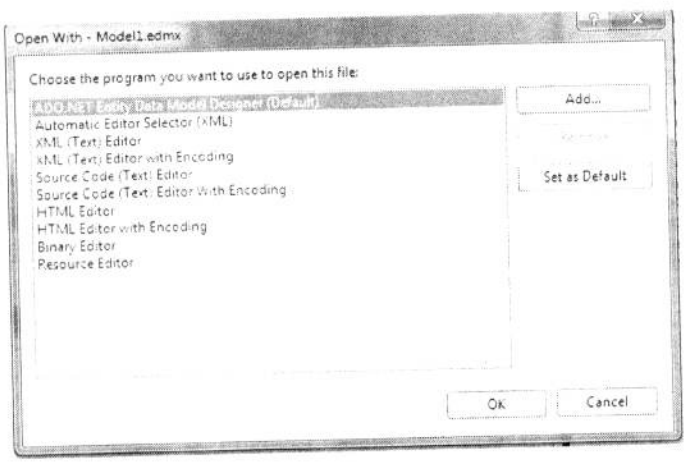


Figura 1.10.- Selección de un editor para un archivo edmx

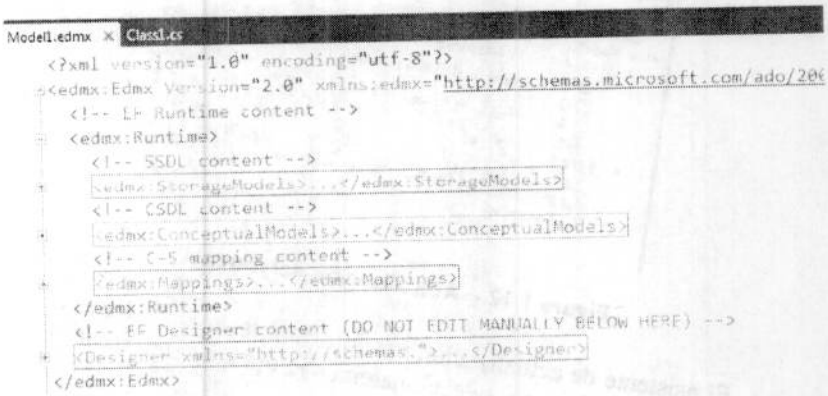


Figura 1.11.- Vista XML de un archivo EDMX

3. Asistente de actualización de modelos

Esta herramienta se utiliza para actualizar el modelo EDM después de que se hayan realizado cambios en la base de datos con la que estemos trabajando. Para usar esta herramienta, basta con situarse en el Navegador del modelo y seleccionar de su menú contextual la opción "Actualizar modelo desde la base de datos". El asistente de actualización de modelos pone a nuestra disposición tres pestañas de trabajo:

Tabla 3.- Opciones de actualización de modelos

Opción	Descripción
Agregar	Permite incorporar al modelo nuevos elementos del esquema de la base de datos.
Refrescar	Muestra los elementos actuales en el modelo y permite seleccionar aquellos cuya definición queramos refrescar o actualizar.
Eliminar	Muestra los elementos actuales en el modelo y permite seleccionar cuales no deseamos que pertenezcan a él.

A continuación mostramos un sencillo ejemplo de uso del asistente de actualización de un modelo EDM. Por ahora, lo único que haremos será agregar una nueva columna a la tabla *Customer* con el fin de especificar su número de pasaporte, *PassportNumber*. Una vez agregada la columna en el modelo relacional con el que estamos trabajando, seleccionaremos la ventana de navegación del modelo y haremos clic en la opción de

actualización del modelo conceptual que tenemos en su menú contextual (figura 1.12).

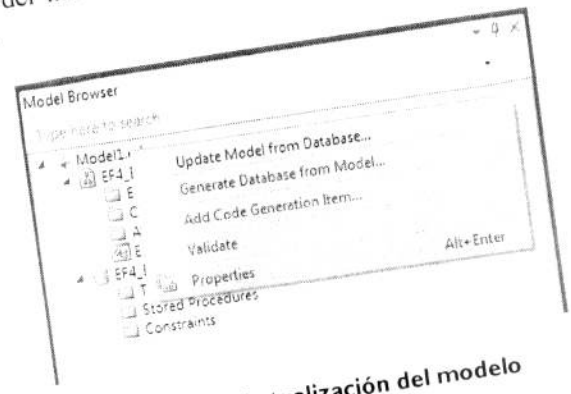


Figura 1.12.- Actualización del modelo

El asistente de actualización nos mostrará las tres opciones de la tabla anterior; en nuestro caso, seleccionaremos la opción "Refrescar el modelo" (figura 1.13).

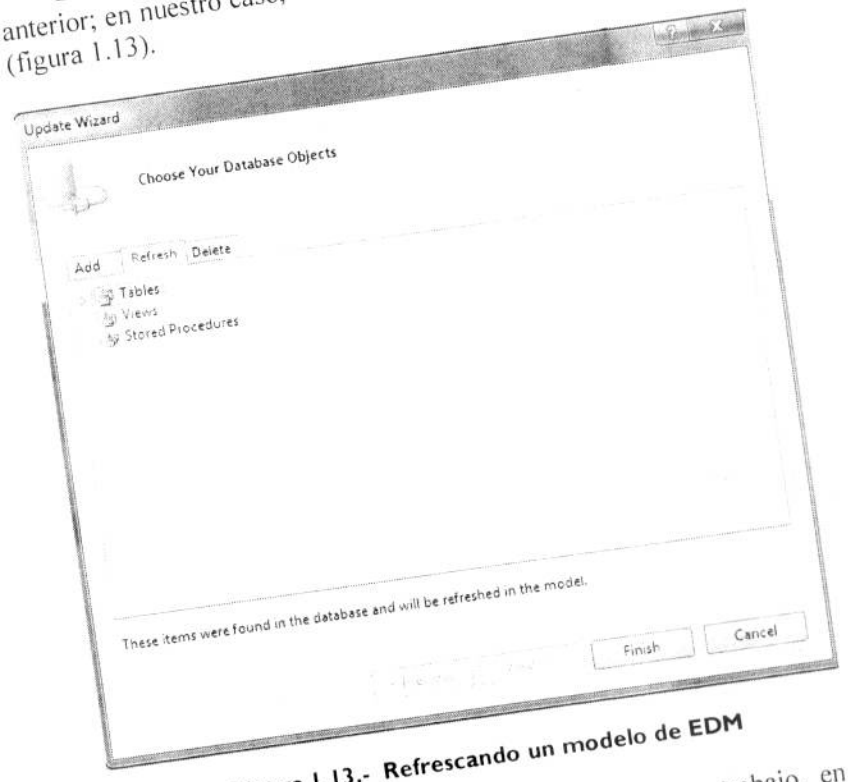


Figura 1.13.- Refrescando un modelo de EDM

Una vez que el asistente haya terminado su trabajo, en este caso agregar una nueva propiedad a la entidad *Customer*, podremos ver como el diseñador nos actualiza la información (figura 1.14).

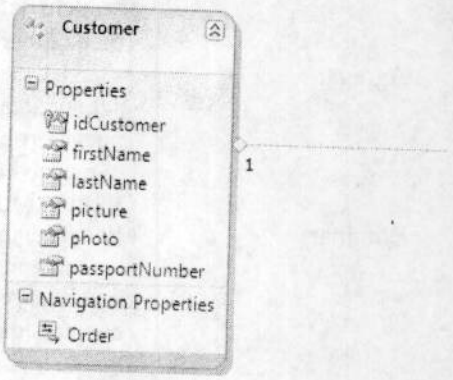


Figura 1.14.- Entidad después de actualizar

Una vez que se produce la actualización de un modelo, se sobrescribe la sección SSDL del fichero edmx correspondiente, por lo que si previamente hubiéramos realizado cualquier modificación sobre él, tales cambios se perderían y no se verían reflejados en el nuevo modelo. Sobre las secciones CSDL y MSL el asistente de actualizaciones solamente permite agregar elementos. Por ello cuando se eliminen tablas y/o asociaciones de la base de datos subyacente, éstas seguirán presentes en el modelo EDM y deberá ser usted manualmente quien los elimine. En la siguiente tabla se puede ver una lista de los cambios que se producen en los modelos EDM en función de las secciones realizadas en la base de datos subyacente.

Tabla 4.- Cambios en el modelo en función de las acciones

Objeto	Cambio	Resultado en el modelo EDM
Tabla/Vista	Agregar	Se agrega una nueva entidad al modelo con sus correspondientes asociaciones y mapeos.
	Eliminar	La tabla o vista se elimina de la sección SSDL y sus datos de mapeo también son eliminados.
	Renombrar	La entidad se renombra y los datos de mapeo se actualizan.
	Inserción de clave primaria	Si una nueva clave primaria es agregada a una tabla, el proceso de actualización incluye una nueva propiedad clave en la entidad. Si ésta forma parte de alguna relación, también se actualizarán

		las propiedades de navegación y asociaciones.
	Eliminación de clave primaria	Si una columna se elimina como miembro de la clave primaria de una tabla, la propiedad correspondiente a ésta en el modelo deberá ser eliminada manualmente.
Columna	Agregar	El asistente de actualización actualiza la entidad agregando una nueva propiedad y estableciendo el mapeo correspondiente en la sección MSL.
	Eliminar	El asistente de actualización únicamente elimina el mapeo de la sección MSL; deberemos ser nosotros quienes eliminemos manualmente la propiedad de la entidad.
	Renombrar la columna	Renombrar la columna tiene el mismo efecto que eliminar dicha columna y volverla a agregar.
	Cambios de la definición	Los cambios de la definición de una columna (por ejemplo, de NVARCHAR(20) a NVARCHAR(50) no provocan ningún cambio o actualización en el modelo. Deberemos ser nosotros quienes realicemos los cambios oportunos.

4. **EDMGen o Generador de EDM**

Por último, **EDMGen.exe** es una herramienta de línea de comandos usada para trabajar con modelos EDM. Los principales usos que se le pueden dar a EdmGen son los siguientes:

- Conectar con una fuente de datos usando un proveedor específico de ADO.NET EF y generar las secciones SSDL, CSDL y MSL de EDM.
- Validar un modelo existente.
- Producir código fuente a partir de la sección CSDL.

Por defecto, EdmGen se sitúa en el directorio de la instalación de .NET Framework 4.1, por lo que es accesible directamente desde la ventana de comandos de Visual Studio 2010. Aunque podría pensarse que en la mayoría de los casos esta herramienta no va ser utilizada puesto que la tenemos perfectamente integrada dentro del entorno de desarrollo (Visual Studio 2010), es necesario conocer su existencia y posibles usos, pues podría ser conveniente utilizarla en algún proceso avanzado de generación automática de código basado en modelos de entidades. En la siguiente tabla se muestra una lista de los diferentes modos de trabajo que ofrece EdmGen.

Tabla 5.- Modos de trabajo de EdmGen

Modo	Descripción
/mode:ValidateArtifacts	Valida los archivos SSDL, CSDL y MSL, mostrando por pantalla los errores y advertencias encontrados.
/mode:FullGeneration	Genera los archivos SSDL, CSDL y MSL usando la información de la conexión a la base de datos especificada en la opción *connectionstring*, y además los ficheros de código fuente correspondiente.
/mode:FromSSDLGeneration	Permite generar los archivos CSDL,MSL y el código fuente a partir de un archivo SSDL.
/mode:EntityClassGeneration	Genera código fuente con las clases correspondientes a las entidades a partir de la información en un archivo CSDL.
/mode:ViewGeneration	Genera código fuente con vistas de cliente generadas a partir de los archivos SDDL, CSDL y MSL.

Junto con estos modelos de trabajo, se requiere el uso de algunas otras opciones de la línea de comandos. Encontrará toda la información relacionada con la utilización de EdmGen en el Apéndice C.

1.2.4.- Entity Client

Entity Client (o EntityClient, unido, nombre con una connotación más cercana al código) es sin duda alguna, junto a EDM, uno de los elementos centrales dentro de ADO.NET Entity Framework. Como seguramente sabrá, desde la versión 2.0 de .NET disponemos de jerarquías de clases que engloban bajo un mismo "paraguas de herencia" las clases necesarias para trabajar con los proveedores específicos para cada motor de base de datos, como SqlClient u OracleClient. Entity Client representa un nuevo conjunto de clases que se integran armónicamente dentro de esas jerarquías.

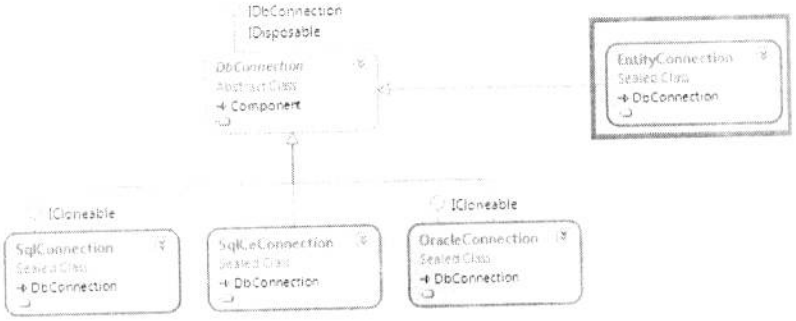


Figura 1.15.- Jerarquía de clases DbConnection

Entity Client es un nuevo proveedor de ADO.NET cuya principal diferencia con el resto de proveedores radica en que, a diferencia de los proveedores tradicionales que trabajan con los modelos físicos de sus bases de datos, con Entity Client consultaremos los modelos de EDM, por lo que aprovecharemos en todo momento el agnosticismo de la base de datos subyacente. Entity Client implementa también a su nivel una arquitectura abierta, y es capaz de trabajar con diferentes proveedores de EF específicos, que son quienes se encargan de traducir las consultas sobre el modelo en consultas en el dialecto específico de la base de datos subyacente, así como de la ejecución de dichas consultas y la recuperación de los resultados (para lo que probablemente se apoyen en otros proveedores de ADO.NET).

El lenguaje que se utiliza para consultar los modelos de EDM se llama **Entity SQL** (abreviadamente, eSQL), y es una variante de los dialectos SQL tradicionales que mejora algunas aspectos de las consultas, como las navegaciones entre tablas. En el siguiente capítulo hablaremos profundamente sobre eSLQ y veremos múltiples ejemplos de consultas en este lenguaje. Por el momento, para que se vaya formando una idea, le ofrecemos un fragmento de código en el que se utilizan las principales clases de Entity Client para ejecutar una consulta eSQL.

```
//Creamos la conexión contra el modelo EDM
    using (EntityConnection connection =
        new EntityConnection(connectionString))
    {
        //Creamos un comando
        using (EntityCommand command = connection.CreateCommand())
        {
            command.Connection = connection;
            //Establecemos la consulta en eSQL
            command.CommandText =
                "SELECT VALUE c FROM XXXEntities.Customer as c";

            //Abrimos la conexión
            connection.Open();

            //Ejecutamos la consulta
            using (EntityDataReader reader =
```

```
                command.ExecuteReader(CommandBehavior.SequentialAccess))
                {
                    while (reader.Read())
                    {
                        //TODO: leer, materializar los resultados aquí
                    }
                }
            }
        }
```

Si usted ha trabajado anteriormente con ADO.NET 2.0 (modelo de acceso a datos clásico en .NET), seguramente este fragmento de código le será tremendamente familiar.

1.2.5.- Object Services

Acabamos de ver como con Entity Client y eSQL realmente podemos cubrir la mayoría de las necesidades de una capa de acceso a datos. Sin embargo, este modelo es el mismo del que disponíamos hasta este momento: por supuesto, con las ventajas de la consulta de modelos conceptuales y el trabajo con múltiples bases de datos. Pero seguiremos sufriendo el trabajo de la recuperación de los resultados obtenidos de la capa de datos para transformarlos en objetos de las distintas clases que tengamos en el conjunto de clases que confirman nuestro dominio del problema (proceso que se conoce como **materialización**). Para salvar este desajuste, EF ofrece **Object Services,** un conjunto de clases que, entre otras facilidades, nos permiten obtener directamente objetos materializados a partir de las consultas eSQL que ejecutemos. Al mismo tiempo que disponemos de una materialización automática de los resultados, ganaremos también todas las ayudas que el compilador y el entorno integrado pueden ofrecernos, al estar trabajando en todo momento de una forma fuertemente tipada.

En contraposición al ejemplo del punto anterior, en el que se consultaban datos de un modelo conceptual usando Entity Client, a continuación podremos ver cómo obtener esos mismos datos usando Object Services y materializando de forma automática los resultados.

```
using (MAmazonEntities dbContext = new MAmazonEntities())
{
    List<Cliente> clientes = dbContext.
        CreateQuery<Cliente>(
            "SELECT VALUE c FROM MamazonEntities.Cliente as c").
        ToList();
}
```

1.2.6.- LINQ To Entities, L2E

Llegados a este punto, ya hemos visto varias de las múltiples ventajas que nos ofrece EF, incluyendo la posibilidad de realizar consultas sobre modelos conceptuales y la

materialización automática de objetos a partir de los resultados de esas consultas. Sin embargo, si preguntásemos a programadores de aplicaciones centradas en datos cuál es la mayor fuente de errores que sufren durante sus horas de desarrollo, es casi seguro que uno de los elementos que mencionarían serían los errores en la escritura de las consultas que crean. Puesto que estas consultas para el compilador no son más que meras secuencias de caracteres, éste no podrá validarlas ni ayudarnos en la codificación de las mismas, de forma que, aun cuando una consulta no esté bien formada o sea incorrecta, podremos compilar y ejecutar, quedando la detección de posibles errores para tiempo de ejecución. La llegada de LINQ abrió una vía increíble para salvar este inconveniente, sentando las bases para la utilización directa de consultas integradas e nuestro lenguaje de programación. Y por supuesto, Microsoft no podía sacar a la luz una nueva tecnología de acceso a datos sin que ésta estuviera "habilitada para LINQ".

El "sabor" de LINQ que hace posible las consultas integradas sobre los modelos de entidades se llama precisamente **LINQ to Entities (L2E)**, y ha sido construido sobre la base de Object Services. A continuación puede ver cómo realizar la consulta presentada anteriormente con L2E.

```
using (XXXEntities dbContext = new XXXEntities())
{
    IQueryable<Cliente> clientes = from c in dbContext.Clientes
                                   select c;
}
```

CAPÍTULO

Entity Data Model

1.- INTRODUCCIÓN

A lo largo del anterior capítulo se realizó una pequeña introducción al significado y la importancia de los modelos de dominio, además de dejar bien claro como las entidades y la propia definición de las mismas juegan un papel muy importante en una solución. En ese mismo capítulo, en la segunda parte, dentro de los distintos componentes de la arquitectura de **ADO.NET Entity Framework**, se habló sobre **Entity Data Model** como la pieza que nos permitirá desarrollar y crear modelos de entidades para nuestros dominios. A lo largo de este capítulo, profundizaremos en EDM y veremos las distintas posibilidades que éste nos ofrece así como escenarios comunes cuando nos enfrentamos a la creación de nuestro primer modelo. Con el fin de simplificar la lectura y facilitar la comprensión se realizarán casos prácticos de aquellas secciones más importantes.

Nota: Tal y como se explicó en el capítulo anterior ADO.NET Entity Framework 4 nos proporciona la posibilidad de crear modelos de EDM partiendo de una base de datos ya existente o bien empezar diseñando el modelo y posteriormente generar un esquema de base de datos acorde con el mismo. En el argot a este tipo de forma de trabajar se la conoce como 'Model First'. Puesto que el objetivo principal de este capítulo es el de ayudar a construir un modelo de entidades y estudiar sus distintas posibilidades nos decantaremos por la primera de las opciones, es decir, partir de un esquema de base de datos dado.

1.1.- Elementos fundamentales de EDM

Sin lugar a dudas, las entidades y las relaciones son los elementos fundamentales dentro de **EDM**. Las entidades, definidas dentro de los modelos mediante el concepto de **EntityType**, suelen estar compuestas por una o más propiedades. Algunas de estas propiedades (las de tipo clave) permiten identificar de forma unívoca e inequívoca las instancias de estas entidades dentro de una determinada colección de ellas. Como usted ya sabrá una vez llegado a este punto, la definición de las entidades en un modelo *EDM* no es más que el comienzo de la definición de las entidades en un modelo de dominio. Pero un error que muchos desarrolladores suelen cometer es el de asumir la correspondencia directa entre las entidades de un modelo de *EDM* y las entidades de un modelo de dominio, algo que no siempre tiene que ser cierto. Piense que *EDM* es una abstracción del modelo relacional y una tabla de una base de datos podría dar por ejemplo lugar a varias entidades si utilizamos algunas de las técnicas de modelado (se verán posteriormente), y una de estas entidades podría entenderse como un **Value Object** y no una entidad, un buen ejemplo de ello ya se puso de manifiesto en la Figura 1.1.

Nota: Un modelo puede tener un número indefinido de objetos (entidades y objetos-valor), y normalmente estarán relacionados entre ellos, incluso de formas complejas. Es decir, un mismo objeto entidad puede estar relacionado con varias entidades, no solo con otra entidad. Así pues, **uno de los principales objetivos que debemos tener es poder simplificar al máximo el número de relaciones presentes en el modelo de entidades del dominio**. Para esto aparece el concepto o **patrón Aggregate Root.** Un agregado es un grupo/conjunto de objetos asociados que se consideran como una única unidad en lo relativo a cambios de datos. El agregado se delimita por una frontera que separa los objetos internos de los objetos externos. Cada agregado tendrá un objeto raíz que será la entidad raíz y será el único objeto accesible, de forma inicial, desde el exterior. El objeto entidad raíz tendrá referencias a cualquiera de los objetos que componen el agregado, pero un objeto externo solo puede tener referencias al objeto-entidad raíz. Si dentro de la frontera del agregado hay otras entidades (también podrían ser también 'objetos-valor'), la identidad de esos objetos-entidad es solo local y tienen solamente sentido perteneciendo a dicho agregado y no de forma aislada

En realidad, dentro de la definición de modelos de dominio, existen algunos conceptos que por ahora no es posible representar en EDM de forma directa, como por ejemplo el concepto de agregado de entidades o **Aggregate Root**, si bien algunas extensiones [4]podrían ayudarnos a implementarlos.

[4] **http://blogs.msdn.com/b/dsimmons/archive/2010/07/12/entityroots-an-ef-extensibility-exercise.aspx**

En EDM además del concepto de **EntityType** para representa a cada una de las entidades, se define también el de **EntitySet**, entendiendo este último como la raíz de unos tipos determinados. Es decir, imagine que en su modelo se definen las entidades *Persona*, *Cliente* y *Empleado*, con una herencia, siendo lógicamente Cliente y Empleado subclases de Persona. En este escenario EDM define como EntitySet a *Persona* puesto que es la raíz de una jerarquía, y *Cliente* y *Empleado* son EntityType.

De forma lógica si en un modelo no hay relaciones de herencia todos los EntityType son a su vez EntitySet. Para terminar, *Entity Data Model* incluye los distintos EntityType y EntitySet de un modelo en un contenedor que recibe el nombre de **EntityContainer**.

1.1.1.- Caso práctico: Las entidades

Bien, mucha literatura seguida, ¿verdad?. Vamos a intentar ir a la práctica que es lo que nos gusta a los desarrolladores, o sea que, si está leyendo sentado en un sillón o tumbado en la cama levántese e inicie su sesión, que vamos a abrir Visual Studio 2010.

Nota: Como ya se comentó en una nota anterior aunque EDM permite la creación de modelos sin necesidad de partir de una base de datos, **Model First**, nosotros trabajaremos siempre partiendo de un esquema determinado. En los ejemplos, para facilitar la lectura, pondremos el código **DDL** que se usará en cada uno de ellos. A disposición del lector en la página web de la editorial encontrará los materiales de cada uno de los ejercicios. Distribuidos entre las carpetas *Code* y *Sql* podrá encontrar la solución de Visual Studio y el código T-SQL necesario para la ejecución de los mismos.

Para nuestro primer ejemplo partiremos de una base de datos cualquiera con un esquema igual que el que se puede observar a continuación.

```
--SQL\EF4_EX_1.sql

USE [EF4_EX_1]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Customer](
    [idCustomer] [int] NOT NULL,
    [firstName] [nvarchar](50) NOT NULL,
    [lastName] [nvarchar](50) NOT NULL,
    [age] [int] NOT NULL,
    [zipcode] [nvarchar](10) NOT NULL,
    [street] [nvarchar](200) NOT NULL,
    [city] [nvarchar](50) NOT NULL,
CONSTRAINT [PK_Customer] PRIMARY KEY CLUSTERED
(
    [idCustomer] ASC
```

```
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO
```

Por supuesto, amigo lector, no pretendemos enseñar a crear modelos relacionales, y los ejemplos suelen estar forzados para explicar y llevar al lector a nuevos escenarios, de ahí que, le ruego no sea demasiado crítico con las estructuras relacionales de los ejemplos.

Una vez que ya disponemos de la base de datos con la estructura anterior, crearemos un nuevo proyecto de Consola en Visual Studio 2010 al que llamaremos EF4_EX_1. En este proyecto, agregaremos un nuevo elemento, un modelo de entidades de ADO.NET, Figura 2.1, llamado EX1Model.edmx.



Figura 2.1.- Creación de un modelo de entidades de ADO.NET EF 4.1

Ahora que ya hemos creado el modelo, seguiremos los pasos del asistente de selección de base de datos. Tiene un ejemplo de ello en el primer capítulo del libro. Terminado el asistente, éste nos da la posibilidad de seleccionar qué elementos deseamos incluir dentro de nuestro modelo de *EDM*. Por ahora, solamente nos fijaremos en las tablas y seleccionaremos *'Customers'* antes de finalizar el asistente.

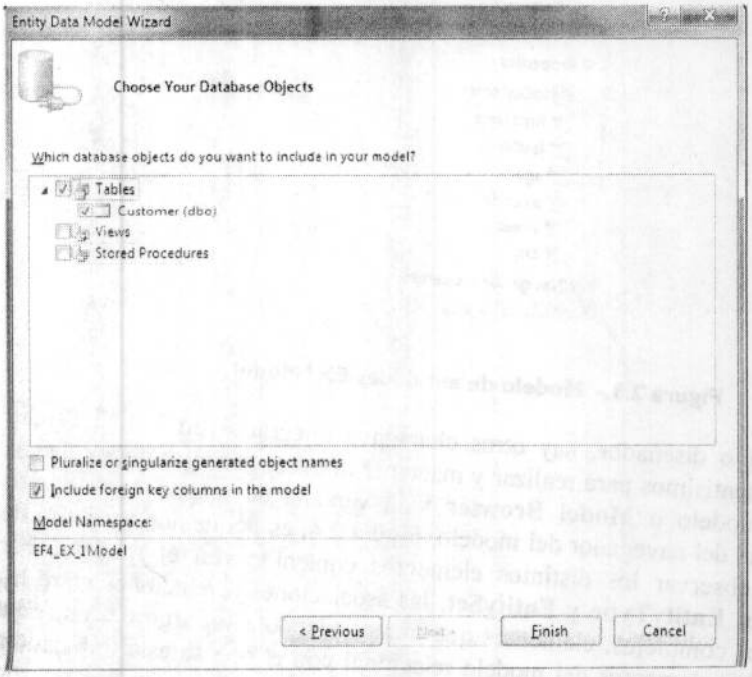


Figura 2.2.- Creación de un modelo de entidades de ADO.NET EF 4.1

La opción 'Pluralizar o singularizar los nombres de los objetos generados' nos permite indicar al asistente que, de forma independiente a como estén los nombres de los objetos de la base de datos -tablas en este caso-, al nombre del *EntityType* lo singularice y al *EntitySet* lo pluralice. Es decir, que el *EntityType* sea *Customer* para nuestro ejemplo y el *EntitySet* sea *Customers*.

Nota: La pluralización y la singularización solamente funciona en lengua inglesa, es decir, se aplicarán las normas de la gramática inglesa para singularizar y pluralizar. Debido a ello, si intenta singularizar/pluralizar elementos nombrados en castellano se puede encontrar con que el *EntitySet* de una tabla Producto sea Productoes.

La opción las columnas de clave externa dentro del modelo será discutida más adelante, previamente es necesario introducir al lector en los diferentes tipos de relaciones que podemos construir en *ADO.NET Entity Data Model*, por lo que por el momento la dejaremos sin seleccionar.

Una vez finalizado el asistente de creación del modelo, al que hemos llamado EX1Model, nos encontraremos con nuestro diseñador de entidades, lógicamente con sólo una entidad ya que, por defecto, el asistente nos da un reflejo fiel de lo que disponemos dentro de nuestro esquema relacional.

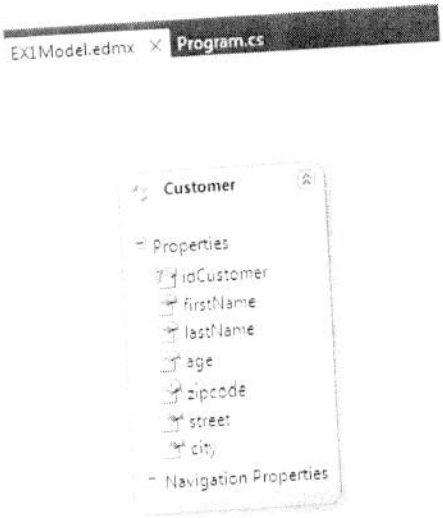


Figura 2.3.- Modelo de entidades EX1Model

Además del propio diseñador, hay otros elementos integrados en Visual Studio 2010 que son importantísimos para realizar y mantener nuestros modelos de entidades. El navegador del modelo o **Model Browser** y la ventana de mapeo son algunos ejemplos. La ventana del navegador del modelo, Figura 2.4, es la que nos permitirá de un vistazo rápido observar los distintos elementos contenidos en él modelo. Por ejemplo los distintos **EntityType** y **EntitySet**, las asociaciones o relaciones entre las entidades y los tipos complejos, elementos que iremos viendo a lo largo de todo este capítulo. También los elementos del modelo relacional con el que se está trabajando, tablas, vistas, restricciones, procedimientos almacenados, etc.

Para mostrar esta ventana solamente tenemos que seleccionar la opción correspondiente en el menú contextual de nuestro diseñador *EDM*, Figura 2.5.



Figura 2.4.- *El Navegador del modelo*

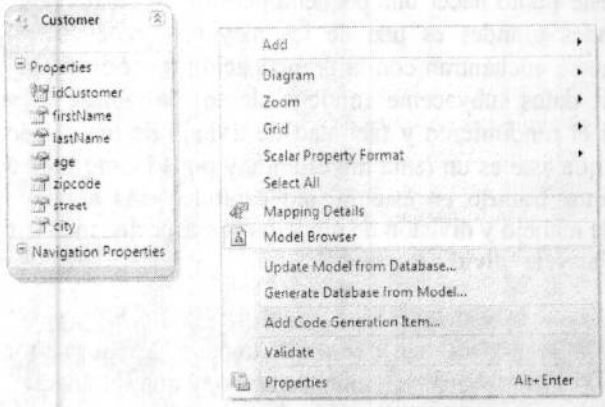


Figura 2.5.- Menú contextual del diseñador de entidades

Dentro de las diferentes opciones que tenemos en la ventana del navegador del modelo se encuentra por ejemplo la posibilidad de actualizar nuestro modelo de entidades, volviendo a seleccionar de la base de datos aquellos elementos que queremos incorporar, borrar o bien actualizar, por medio de la opción "actualizar modelo a partir de la base de datos", ver Figura 2.6.

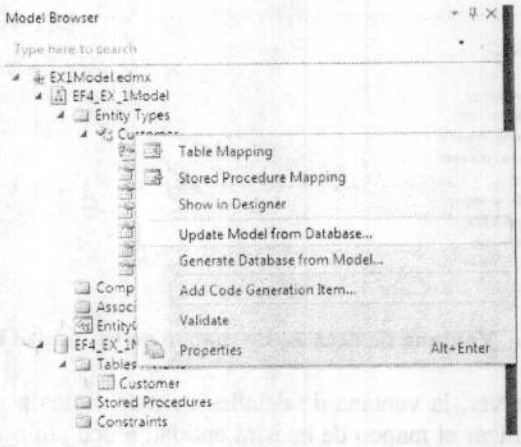


Figura 2.6.- Actualizar el modelo de entidades desde el navegador del modelo

Nota: Seguramente, estimado lector, se habrá dado cuenta de que la opción de actualizar el modelo a partir de la base de datos también está presente dentro del menú contextual del diseñador, tal y como se puede apreciar en la Figura 2.5.

Además de esta opción que podemos tomar en el navegador del modelo, también podremos hacer búsquedas y hacer que los elementos seleccionados se resalten en el diseñador, opciones nuevas en Visual Studio 2010, con el fin de facilitar más si cabe el trabajo de los desarrolladores con modelos de entidades grandes.

Permítame en este punto hacer una pequeña pausa para resaltar que el trabajo con modelos de entidades grandes es una de las mayores preocupaciones de muchos desarrolladores, que se encuentran con la preocupación de cómo trabajar cuando, por ejemplo, la base de datos subyacente contiene cientos de tablas. Y por supuesto les preocupa cuál será el rendimiento y facilidad de trabajo de los diseñadores en estos escenarios. Puesto que este es un tema importante, y por lo tanto merece una mención especial, intentaremos tratarlo en este mismo capítulo, más adelante, juntando los aspectos técnicos de manejo y división de modelos con aspectos más teóricos en cuanto a la forma de trabajar y de 'dividir lo grande'.

Nota: Dividir lo grande es una expresión tomada del libro **Domain Driven Design de Eric Evans**, del cual ya hemos hablado, y que retomaremos tal y como acabamos de comentar en el último párrafo cuando se traten los problemas y formas de a abordar la construcción de modelos de entidades a partir de modelos relacionales con ingentes cantidades de vistas y tablas.

La última de las acciones que comentaremos por ahora (más adelante se verán otras opciones) tiene que ver con otro de los elementos de vital importancia en la creación de modelos de entidades: la acción de mostrar la ventana de **detalles de mapeo** de las entidades seleccionadas en la ventana de navegación del modelo, Figura 2.7.

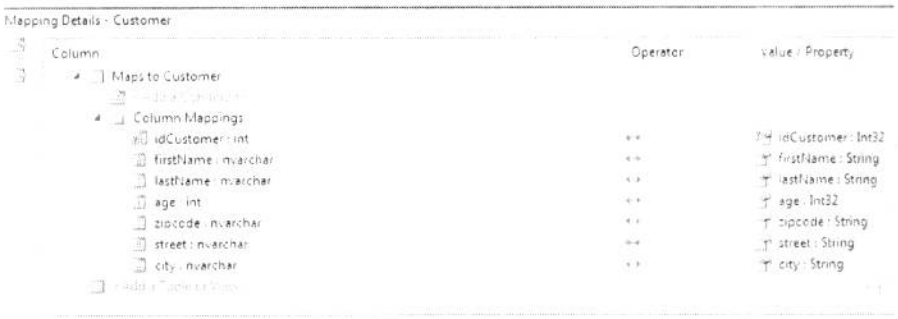


Figura 2.7.- Ventana de detalles de mapeo de la entidad Customer

Como podrá observar, la ventana de detalles de mapeo nos permite, de una forma visual e intuitiva, realizar el mapeo de nuestra entidad a una tabla y las propiedades de nuestra entidad a las columnas de una tabla. Por supuesto, estamos en el escenario más simple, pronto verá como una misma entidad puede estar asociada a varias tablas o bien una tabla mapearse contra dos entidades, así como otras posibilidades avanzadas del diseñador.

Como se podrá imaginar, cada una de las propiedades tiene que tener una correspondencia correcta con los tipos de las columnas de la tabla. Si por ejemplo, intentásemos mapear la propiedad *age* de nuestra clase con la columna *city* de la tabla *Customers*, Figura 2.8, al compilar la validación del modelo nos mostraría un error similar al siguiente.

```
    Error 1     Error 2019:
    Member Mapping specified is not valid. The type
'Edm.String[Nullable=False,DefaultValue=,MaxLength=50,Unicode=True,Fixed
Length=False]' of member 'city' in type 'EF4_EX_1Model.Customer' is not
compatible with 'SqlServer.int[Nullable=False,DefaultValue=]' of member
'age' in type 'EF4_EX_1Model.Store.Customer'.
    C:\XXX\EF4_EX_1\EX1Model.edmx     50      13      EF4_EX_1
```

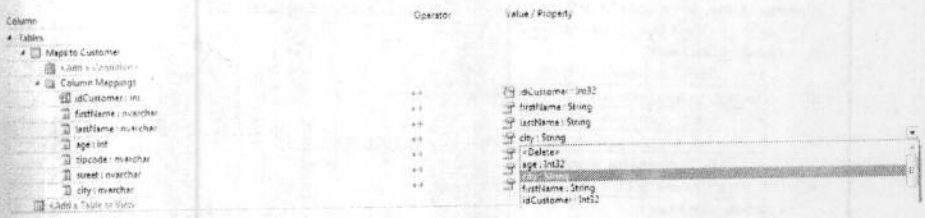


Figura 2.8.- Mapeo incorrecto de una propiedad

Lógicamente, además del tipo, las propiedades de nuestras entidades tienen más atributos que podemos configurar y adaptar. Éstos se pueden observar seleccionando una propiedad de la entidad y mostrando la 'Ventana de Propiedades' de Visual Studio (pulsando F4).

De entre los atributos más importantes de una entidad, podemos resaltar los siguientes, ver Tabla 1.

Tabla 1.-Propiedades de una entidad

Propiedad	Significado
Type	Nos permite especificar un tipo para esta propiedad. Estos tipos tienen una correspondencia directa con el sistema de tipos .NET.
Setter	El modificador de ámbito que tendrá el método Set de esta propiedad.
Getter	El modificador de ámbito que tendrá el método Set de esta propiedad.
Name	El nombre de la propiedad.
Nullable	Permite establecer si la propiedad admite nulo o no.
StoreGeneratedPattern	Permite determinar si la columna correspondiente en la base de datos debe de ser autogenerada

Entity Key	Permite establecer si esta propiedad es clave de la entidad o no.
Documentation	Permite establecer documentación para esta entidad.
DefaultValue	Nos permite establecer el valor por defecto que se enviará a los comandos de actualización e inserción si no se establece.
ConcurrencyMode	Modo de gestión de la concurrencia

Lógicamente todo este trabajo con el diseñador y las distintas ventanas que nos permiten trabajar en un modelo de entidades tienen su actuación en los distintos espacios de EDM, los cuales ya se comentaron en el primer capítulo de este libro. Estos espacios, se pueden observar más detalladamente si selecciona el fichero del modelo en su ventana de solución de Visual Studio y lo abre con un editor XML, Figura 2.9.

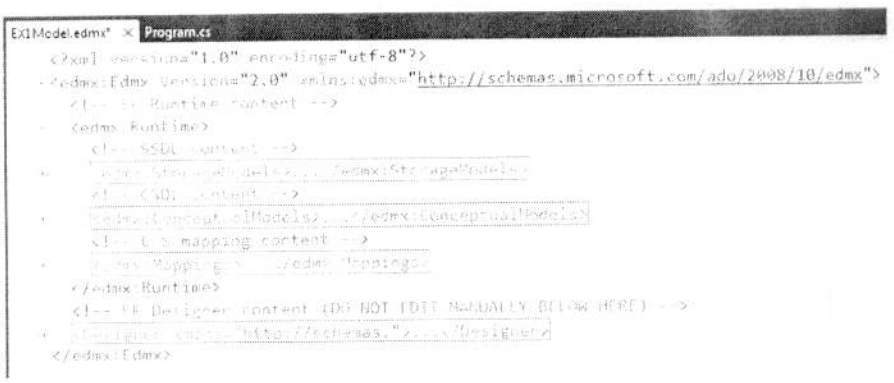


Figura 2.9.- CSDL,SSDL y MSL

Aunque no es habitual (y mucho menos en esta nueva versión de EF), podría encontrarse el caso de necesitar editar manualmente este fichero con el fin de implementar alguna característica no soportada directamente, o bien adaptar algún *issue*. Si esto sucede, valore bien la actuación puesto que en ocasiones editar el fichero de forma manual le impide volver a tener una vista de diseño.

Nota: Algunos casos de necesitar la edición manual de ficheros EDMX pueden venir por el uso de algún proveedor de terceros que no implemente alguna funcionalidad que necesite. Un caso podría ser el de un conocido proveedor para bases de datos Oracle, el cual no nombraremos, que no interpreta correctamente las secuencias y establece la opción de patrón de generación identidad en el espacio SSDL de las entidades.

Tal y como se mostró en la Tabla 1 de este capítulo, uno de los atributos pertenecientes a las propiedades de las entidades es el de *StoreGeneratedPattern* o patrón de generación, el cual nos permite indicar si una propiedad podría ser autogenerada durante el proceso de inserción y/o actualización. De entre los posibles valores que puede tener esta opción se encuentran las de ***None***, ***Identity*** y ***Computed***.

El valor de *Computed* nos permite indicar que esta propiedad debería ser una columna auto-calculada, mientras que *Identity* nos permite indicar que la columna a la que está mapeada es de tipo identidad, si hablamos de Sql Server, o *secuence* si hablamos de Oracle por poner algún ejemplo. En realidad, la opción *Identity* también podría aplicarse a una propiedad aunque esta no tuviera la opción *identity* en el caso de Sql Server.

Veamos un posible ejemplo de esto. Para ello partiremos de una pequeña modificación en el script de nuestra tabla *Customer*, como se muestra a continuación, y crearemos un nuevo proyecto al que llamaremos EF4_EX2 en el que incluiremos un nuevo modelo de entidades, como hemos anteriormente, con esta nueva tabla.

```
--SQL\EF4_EX_2.sql

USE [EF4_EX_1]
GO

SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[CustomerIdentity](
    [idCustomer] [uniqueidentifier] NOT NULL,
    [firstName] [nvarchar](50) NOT NULL,
    [lastName] [nvarchar](50) NOT NULL,
    [age] [int] NOT NULL,
    [zipCode] [nvarchar](10) NOT NULL,
    [street] [nvarchar](200) NOT NULL,
    [city] [nvarchar](50) NOT NULL,
 CONSTRAINT [PK_CustomerIdentity] PRIMARY KEY CLUSTERED
(
    [idCustomer] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]

GO

ALTER TABLE [dbo].[CustomerIdentity] ADD  CONSTRAINT
[DF_CustomerIdentity_idCustomer]  DEFAULT (newid()) FOR [idCustomer]
GO
```

Una vez creado el modelo, si seleccionamos la propiedad *idCustomer* y nos dirigimos a la ventana de propiedades, Figura 2.10, podríamos establecer esta propiedad como *Identity* dentro de *StoreGeneratedPattern*.

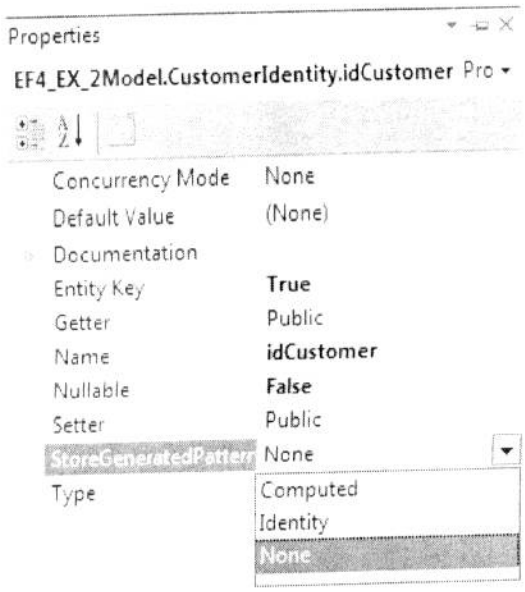


Figura 2.10.- Modificando la propiedad StoreGeneratedPattern

Aunque aún no sabemos nada más de Entity Framework que lo que nos ocupa con respecto a modelado, podríamos pensar que con esta simple acción -establecer *StoreGeneratedPattern* a *Identity*-, a la hora de trabajar realmente en código, EF debería generar un nuevo valor para esta propiedad. Pues bien, si esta forma de pensar fuera la correcta, en realidad nos haría falta un pequeño paso a mayores que consistiría en anotar esta misma opción en el espacio SSDL. Desgraciadamente, esta anotación no la podemos hacer desde el diseñador directamente por lo que podríamos abrir el fichero edmx mediante un editor *xml* y modificar la siguiente línea dentro del espacio SSDL correspondiente a la propiedad *idCustomer* tal y como se ve en los siguientes fragmentos:

```
<!-- SSDL content -->
    …
<Property Name="idCustomer" Type="uniqueidentifier" Nullable="false"
/>
    …
```

```
<!-- SSDL content -->
    …
<Property Name="idCustomer" Type="uniqueidentifier" Nullable="false"
StoreGeneratedPattern="Identity"/>
    …
```

Nota: El trabajo con uniqueidentifier en ADO.NET Entity Framework mezclado con el patrón de generación Identidad tal y como acabamos de ver en el ejemplo EF4_EX_2 podría tener un defecto de rendimiento en EF 4.1, por cómo se hace el proceso de obtención del valor de la columna. Le recomendamos desde aquí, si puede, que evite que estas columnas se autogeneren estableciendo usted el valor de las mismas mediante la sencilla instrucción Guid.NewGuid().

Hasta ahora, ya hemos visto los elementos más importantes de una entidad: sus propiedades, la configuración y las opciones de las mismas. Las propiedades simples o escalares, las que hemos visto hasta el momento, son solamente uno de los tipos de propiedades que podemos tener. Ya desde **Entity Framework 1.0** disponemos de la posibilidad de refactorizar propiedades de nuestras entidades en tipos complejos, aunque, en esta primera versión era algo que no estaba soportado en el diseñador. Ahora, en .NET 4.1 y la nueva versión de EF esta posibilidad ya está incluida directamente en EDM y su uso es de extremada sencillez.

Para mostrar un ejemplo partiremos del modelo creado en el ejemplo EF4_EX_2, Figura 2.11, y seleccionaremos las propiedades *zipCode, Street* y *City*. Una vez seleccionadas veremos como en el menú contextual disponemos de la opción "Refactorizar en un tipo complejo".

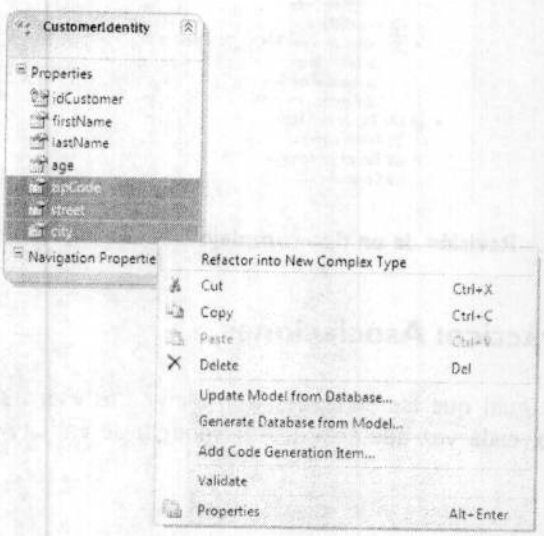


Figura 2.11.- Creación de un tipo complejo

Una vez creado este tipo complejo, al que llamaremos *Address*, podemos ver como la entidad ha sustituido las propiedades mencionadas anteriormente por una nueva propiedad, Figura 2.12, con el nombre *ComplexProperty* por defecto. Lógicamente este nombre se puede modificar cambiando el valor de su atributo ***Name***, que envuelve a las propiedades escalares refactorizadas.

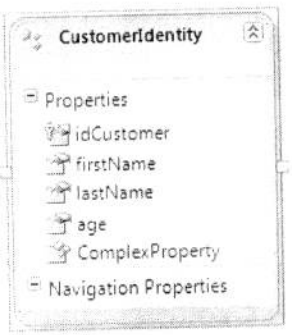


Figura 2.12.- Creación de un tipo complejo

Si deseamos revisar las propiedades que forman el nuevo tipo complejo y/o ver y establecer diferentes atributos de las mismas, solamente tendremos que dirigirnos al navegador del modelo (recuerde esta tarea en unos pasos más atrás) y ver los distintos tipos complejos que hayamos creados, Figura 2.13.

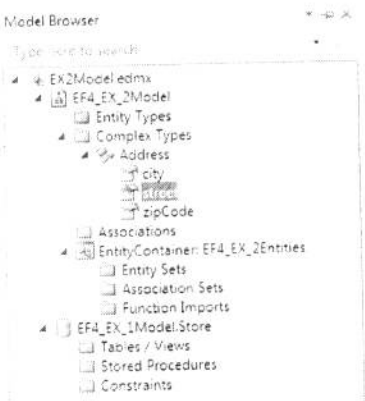


Figura 2.13.- Revisión de un tipo complejo en el navegador del modelo

1.1.2.- Caso Práctico: Asociaciones

Las asociaciones, al igual que las entidades, son elementos esenciales dentro de los modelos. Por defecto, cada vez que creamos un modelo de entidades a partir de una

base de datos, las asociaciones entre las distintas entidades que lo compongan son un fiel reflejo de las relaciones entre las tablas del modelo relacional subyacente. Si con una entidad nuestro modelo creaba el concepto de *EntityType* y *EntitySet*, con las asociaciones dispondremos de forma similar de los elementos *AssociationType* y *AssociationSet*, los cuales, también se podrán manejar y ver desde el navegador del modelo.

Al igual que hemos hecho con las entidades, iremos viendo ejemplos con el fin de introducir al lector en las distintas casuísticas de las asociaciones y en su tratamiento dentro de un modelo de entidades.

Nota: Si recuerda lo tratado en la primera parte del capítulo 1, estará asociando con seguridad el concepto de **Tipo Complejo** con el de **Objeto Valor** o *Value Object*. Aunque esto es cierto, este tipo de refactorizaciones nos dan elementos que no tienen identidad en si mismos. No se quede con el concepto de que solamente los tipos complejos representarán *Value Objects* en un modelo de dominio. En muchas ocasiones, después del diseño de entidades nos encontraremos con que algunas de ellas representan también Objetos Valor.

Para un primer ejemplo partiremos de una base de datos con dos tablas, una de clientes y otra de pedidos. Lógicamente pedidos contiene una clave externa hacia la tabla de clientes. A continuación puede ver el fragmento de *DDL* que nos permite generar el esquema relacional de trabajo.

```
--SQL\EF4_EX_3.sql

USE [EF4_EX]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Customer](
    [idCustomer] [int] IDENTITY(1,1) NOT NULL,
    [firstName] [nvarchar](50) NOT NULL,
    [lastName] [nvarchar](50) NOT NULL,
    [picture] [image] NOT NULL,
 CONSTRAINT [PK_Customre] PRIMARY KEY CLUSTERED
 (
     [idCustomer] ASC
 )WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
 IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
 ON [PRIMARY]
 ) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
 GO

 SET ANSI_NULLS ON
 GO
 SET QUOTED_IDENTIFIER ON
 GO
```

```
CREATE TABLE [dbo].[Order](
    [idOrder] [int] IDENTITY(1,1) NOT NULL,
    [orderDate] [date] NOT NULL,
    [totalOrder] [numeric](12, 2) NOT NULL,
    [idCustomer] [int] NOT NULL,
 CONSTRAINT [PK_Order] PRIMARY KEY CLUSTERED
(
    [idOrder] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO
/****** Object:  ForeignKey [FK_Order_Customer]
ALTER TABLE [dbo].[Order]  WITH CHECK ADD  CONSTRAINT
[FK_Order_Customer] FOREIGN KEY([idCustomer])
REFERENCES [dbo].[Customer] ([idCustomer])
GO
ALTER TABLE [dbo].[Order] CHECK CONSTRAINT [FK_Order_Customer]
GO
```

Una vez que ya tenemos nuestro modelo relacional, si creamos un proyecto de

Nota: Por ahora seguiremos sin seleccionar la opción de inclusión de las columnas de clave externa, 'foreign key columns'.

Visual Studio 2010 que incluya un modelo de entidades de *ADO.NET Entity Framework*, al igual que se hizo en los ejemplos anteriores, podremos ver como *EDM* es capaz de representar las relaciones entre tablas por medio de asociaciones entre nuestras entidades, ver Figura 2.14.

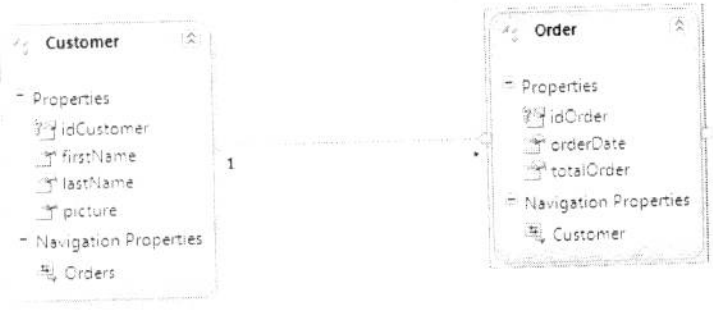


Figura 2.14.- Asociación independiente en EDM

Si se fija en la figura anterior podrá observar cómo, además de las distintas propiedades escalares o complejas (si decide *refactorizarlas*), se han incluido un nuevo tipo de propiedades, conocidas como propiedades de navegación. Las propiedades de navegación, nos permiten ver y alcanzar aquellos elementos que estén asociados con cada entidad. Así, la propiedad *Orders* nos permite ir hacia los pedidos a los que

pertenezca un determinado cliente, y la propiedad de navegación *Customer*, dentro de la entidad *Order*, nos permite ir hacia el cliente asociado a un determinado pedido.

Si echamos un vistazo al navegador del modelo veremos como la asociación creada también formará parte de la información del mismo. Recuerde que en el primer párrafo ya se comentó la existencia de los elementos *AssociationType* y *AssociationSet* (ver Figura 2.15).

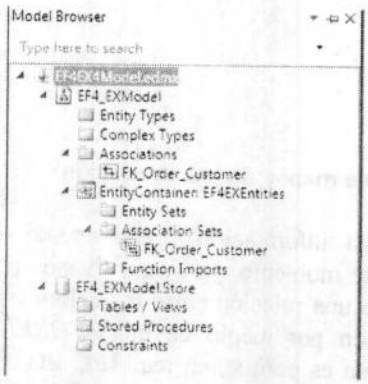


Figura 2.15.- Model browser con una asociaciónFigura

Además, al igual que cualquier otro elemento dentro de EDM, también podemos ver y modificar los distintos atributos de una asociación. Para ello solamente es necesario seleccionarla por medio de un clic y mostrar la ventana de propiedades de Visual Studio 2010.

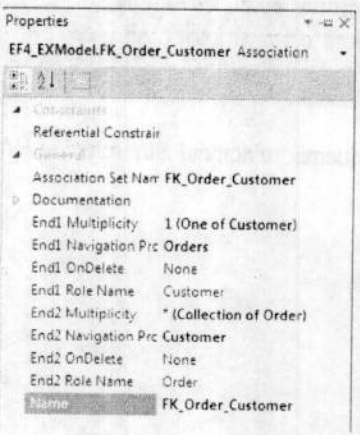


Figura 2.16.- Propiedades de una asociación en EDM

Como habrá podido observar, las propiedades de una asociación definen atributos para cada uno de los 'Roles' o extremos de la misma. La multiplicidad (de la cual seguiremos hablando en los siguientes párrafos), el nombre o bien la acción a tomar en caso de borrado

o eliminación de una entidad son, sin duda, los elementos más importantes dentro de estos atributos. Como se imaginará, la multiplicidad nos permite indicar si el extremo relacionado en una asociación será cero a uno, uno a muchos. Si vemos esto pensando como programadores en un mundo orientado a objetos, básicamente estaremos pensando si la propiedad es de tipo simple o una colección de elementos.

Una de las características más importantes de las asociaciones, tal y como las hemos visto hasta ahora, es que también tienen información de 'mapeo' contra el esquema relacional. Para verlo, solamente hay que seleccionar la asociación y en su menú contextual seleccionar la opción de detalles de mapeo, ver Figura 2.17.

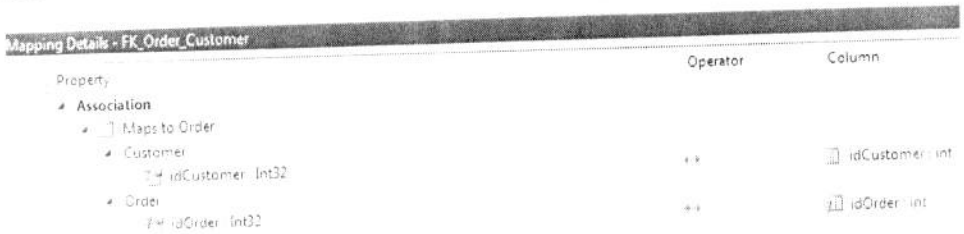


Figura 2.17.- Detalles de mapeo de una asociación

¿Para qué necesitamos especificar esta información? Seguramente, esta sea la pregunta que usted se está haciendo en este momento, ya que, si hemos partido de un modelo relacional en el que había definido una relación entre las columnas de la tabla porque no podemos expresar la asociación por medio de una *'FOREIGN KEY [] REFRENCES'*. La respuesta a esta pregunta es porque, en realidad, esta asociación es independiente de lo que se disponga en el modelo relacional, es decir, podríamos haberla construido aunque en la base de datos no existiera ninguna relación entre las tablas. Por ello, a este tipo de asociaciones se las conoce como **Asociaciones Independientes**.

Nota: Recuerde que la correcta pluralización o singularización de los elementos dependerá de si ha seleccionado esta opción al crear el modelo de entidades y los nombres de los objetos del modelo relacional estén en lengua inglesa. Si no es así, al igual que las propiedades escalares o los tipos complejos también puede renombrar las propiedades de navegación.

Para verlo, partiremos del siguiente esquema relacional sin relación entre las tablas anteriores.

```
--SQL\EF4_EX_4.sql

USE [EF4_EX]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
```

```
CREATE TABLE [dbo].[Customer](
    [idCustomer] [int] IDENTITY(1,1) NOT NULL,
    [firstName] [nvarchar](50) NOT NULL,
    [lastName] [nvarchar](50) NOT NULL,
    [picture] [image] NOT NULL,
 CONSTRAINT [PK_Customre] PRIMARY KEY CLUSTERED
(
    [idCustomer] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Order](
    [idOrder] [int] IDENTITY(1,1) NOT NULL,
    [orderDate] [date] NOT NULL,
    [totalOrder] [numeric](12, 2) NOT NULL,
    [idCustomer] [int] NOT NULL,
 CONSTRAINT [PK_Order] PRIMARY KEY CLUSTERED
(
    [idOrder] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO
/****** Object:  ForeignKey [FK_Order_Customer]
ALTER TABLE [dbo].[Order]  WITH CHECK ADD  CONSTRAINT
[FK_Order_Customer] FOREIGN KEY([idCustomer])
REFERENCES [dbo].[Customer] ([idCustomer])
GO
ALTER TABLE [dbo].[Order] CHECK CONSTRAINT [FK_Order_Customer]
GO
```

Una vez creado la base de datos con el esquema anterior, crearemos un nuevo proyecto de Visual Studio (**code\ef4_ex4)** y un modelo de entidades de *ADO.NET Entity Framework*, ver Figura 2.18.

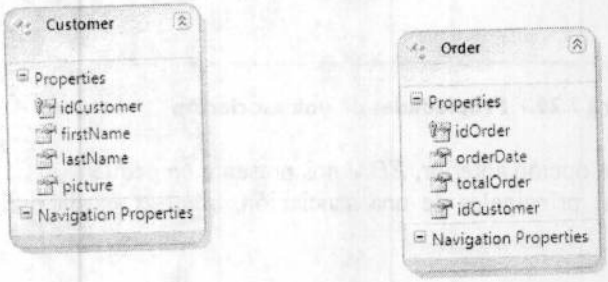


Figura 2.18.- Modelo de entidades sin relaciones

Ahora, que ya tenemos las entidades, solamente queda incluir una asociación tal y como la teníamos en el ejemplo anterior. Para ello podemos ir por dos caminos. El primero sería el seleccionar el elemento *asociación* desde la ventana de herramientas de Visual Studio y posteriormente seleccionar las dos entidades que forman parte de la asociación; en este caso las dos que tenemos. La segunda forma, la que veremos aquí, es seleccionar la entidad *Customer* y, desde su menú contextual, Figura 2.19, seleccionar la opción *"Agregar nueva asociación"*.

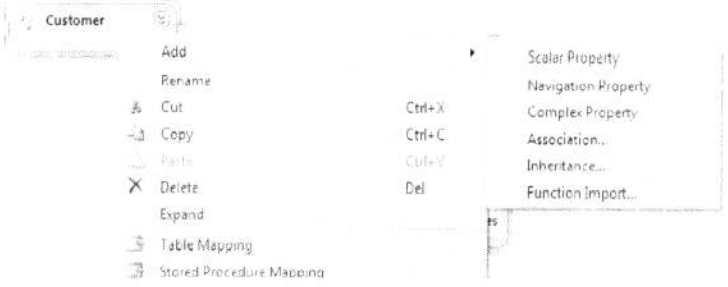


Figura 2.19.- Creación de una nueva asociación

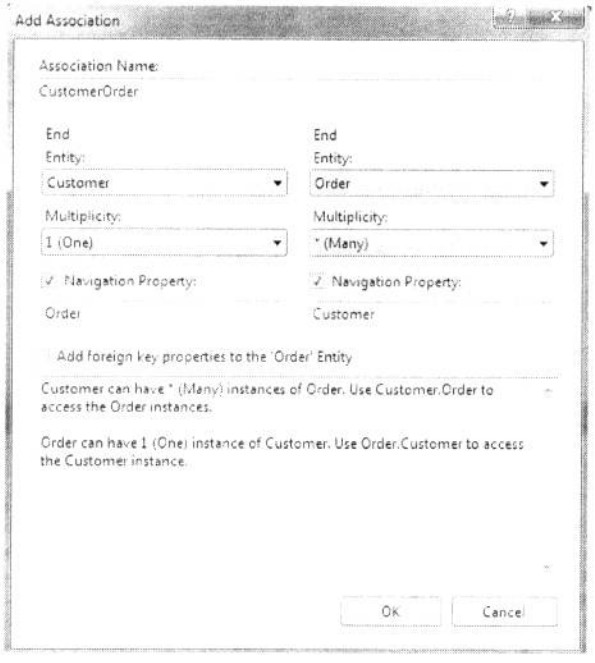


Figura 2.20.- Propiedades de una asociación

Una vez seleccionada la opción anterior, *EDM* nos presenta un pequeño asistente de selección de los elementos principales de una asociación, nombre, cardinalidad etc.

Estas opciones y su significado (ver Figura 2.20) son básicamente las mismas que vimos en la ventana de propiedades de *Visual Studio 2010* en el ejemplo anterior. A mayores también disponemos de una opción de selección para incluir 'columnas de clave externa' que, tal y como hemos hecho hasta ahora, por ahora no seleccionaremos.

Llegados a este punto, solamente queda establecer las condiciones de mapeo, es decir, como indicamos a *Entity Data Model* la relación que esta asociación tiene dentro del modelo relacional. Para ello solamente tenemos que mostrar los detalles de mapeo de la misma, a partir del menú contextual de la asociación, y establecer los valores tal y como se ven en la siguiente figura, 2.21.

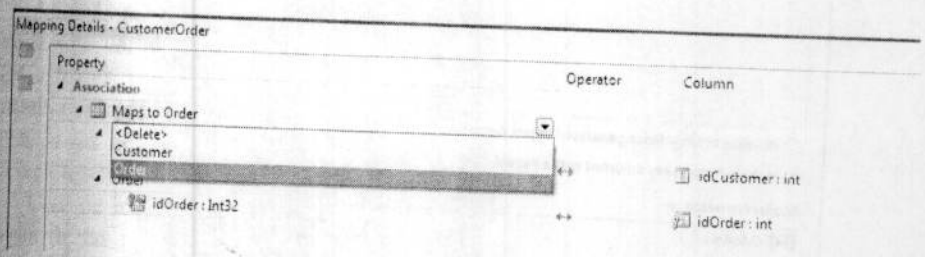


Figura 2.21.- Detalles de mapeo

Como hemos comentado, hasta ahora solamente hemos visto el concepto de 'Asociaciones Independientes', gracias a las cuales en una entidad disponemos de propiedades de navegación que nos permiten movernos por elementos asociados, exista o no en la base de datos una relación entre tablas. Este tipo de asociaciones -las únicas de las que disponíamos en la primera versión de *ADO.NET Entity Framework*- fueron criticadas desde la comunidad de desarrolladores en cierta medida. El porqué, viene del hecho de que dificultaba el trabajo con las asociaciones, puesto que, por ejemplo la actualización del *Cliente* asociado a un *Pedido* implicaba la modificación de una propiedad compleja, la propiedad de navegación Cliente, y no un simple valor escalar como podría ser por ejemplo *idcliente*. Desde el equipo de producto de *ADO.NET Entity Framework*, de forma acertada en mi opinión, decidieron incorporar para esta nueva versión del producto un nuevo tipo de asociaciones que resolvieran las dificultades e inconvenientes que los desarrolladores tenían con respecto al trabajo con las propiedades de navegación. El nuevo tipo de asociaciones incorporadas en *Entity Framework 4.1* se dio a conocer como *FK Associations* o *Asociaciones de clave externa*. El nombre, como se habrá dado cuenta, viene del hecho de que este tipo de asociaciones están ligadas a una relación en la base de datos, sin la cual no podrían definirse.

Con el fin de mostrar un ejemplo, **code\sql_ef4_ex5**, de uso de este tipo nuevo de asociaciones, partiremos de un fragmento relacional anterior, sql\ef4_ex3_sql, en el que se definían las tablas *Customer* y *Order* con una relación entre ellas. Pero ahora, en el último paso del asistente, seleccionaremos la opción *'Incluir columnas de clave externa'*, tal y como podemos ver en la siguiente Figura 2.22.

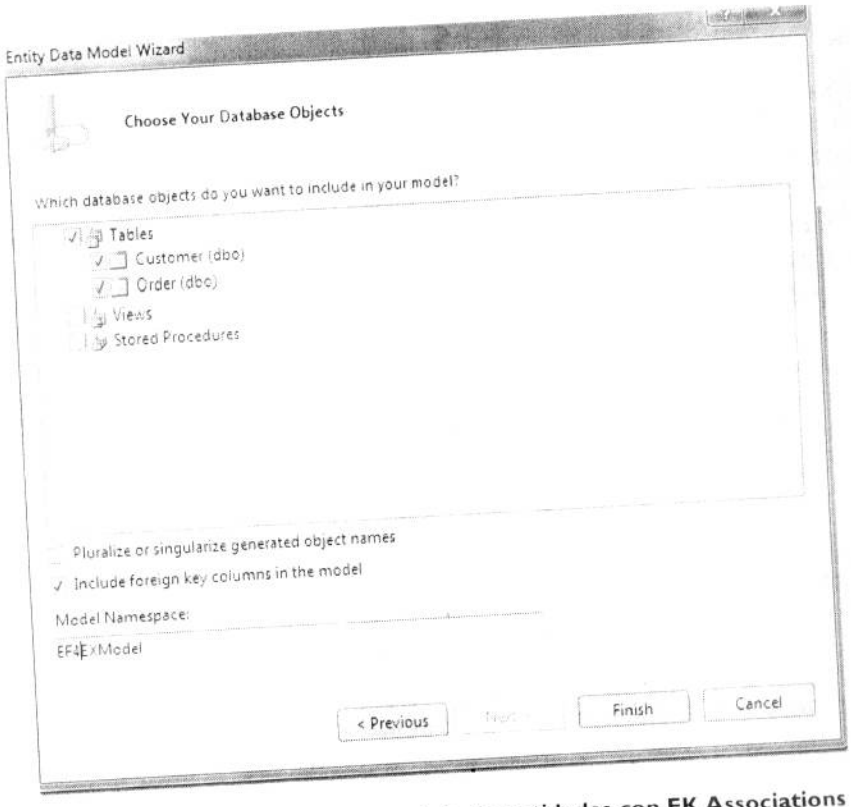


Figura 2.22.- Creación de un modelo de entidades con FK Associations

Aunque en un principio parezca que con esta simple selección dentro del asistente, nada haya cambiado con respecto a las entidades y asociaciones en juego, un vistazo más profundo nos hará cambiar de opinión. En primer lugar, si visitamos la ventana de detalles de mapeo podremos observar como ahora no hay especificación, por nosotros, de cómo se relaciona una asociación con los distintos elementos del modelo, Figura 2.23.

Nota: Dentro de la impedancia entre el mundo orientado a objetos y los modelos relacionales, las relaciones es el típico ejemplo de cambio. Mientras que en los modelos relacionales necesitamos duplicar valores para construir una relación (esta tabla intermedia de la que hablamos en el párrafo anterior es el vivo ejemplo), en el mundo orientado a objetos simplemente necesitamos una referencia.

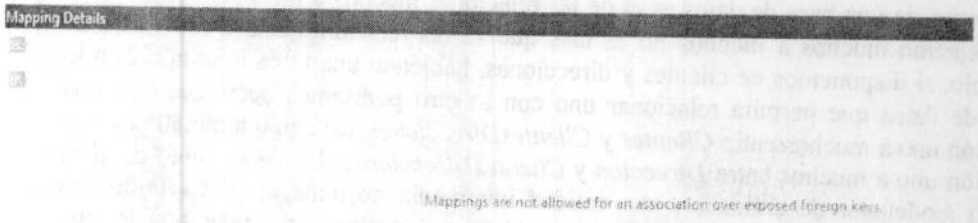


Figura 2.23.- Detalles de mapeo en una FK Association

Además de la imposibilidad de establecer detalles de mapeo en una *FK Association* podemos ver como en la entidad *Order* disponemos de una nueva propiedad, *idCustomer*, que representa el valor de la clave externa, elemento que nos permitiría cambiar rápida y fácilmente una relación (se verá en posteriores capítulos). Esta era una de las demandas más grandes de la comunidad de desarrolladores respecto a las relaciones.

Otro de los elementos que se incluyen con las asociaciones de clave externa es la vista de las restricciones referenciales. Para acceder a la misma, Figura 2.24, puede hacer doble-clic en el elemento asociación del modelo *EDM*, o bien por medio de la ventana de propiedades de Visual Studio de la asociación incluida en el modelo.

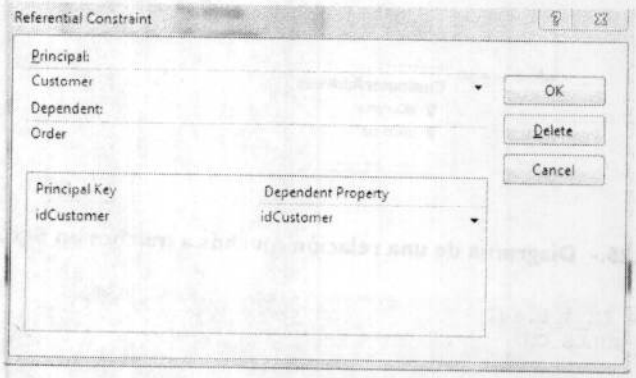


Figura 2.24.- Restricción referencial en una FK Association

Tal y como observará en la figura anterior esta nueva ventana nos permitirá establecer las claves que forman la asociación, así como el nombre de la misma.

A pesar de que las *FK Associations* representan para la mayoría de los desarrolladores un avance y una mejora con respecto a su trabajo con modelos de entidades, no todo son ventajas en este tipo de asociaciones. En primer lugar, las *FK Associations* no son más que parte de la impedancia de los modelos relacionales con respecto a los modelos de objetos, y por lo tanto es una solución de compromiso para un problema no resuelto de forma adecuada para muchas personas que trabajaban con *ADO.NET EF* en su primera versión. Además de este problema, más conceptual que otra cosa, la existencia de *FK Associations* introduce un grado más de complejidad, puesto que disponemos de dos artefactos para resolver un mismo problema.

Una vez presentados los distintos tipos de asociaciones, independientes y basadas en claves externas, veremos a continuación algunos elementos sobre éstas que es conveniente destacar y que el lector conozca.

Hasta ahora hemos hablado de las asociaciones y de los distintos roles o extremos de las mismas, mencionando aunque fuera de pasada, las distintas *cardinalidades* que pudieran tener. Un ejemplo peculiar de cómo *Entity Data Model* interpreta las relaciones de una base de datos es el de las relaciones muchos a muchos. En realidad, una relación muchos a muchos no es más que varias relaciones uno a muchos. Por ejemplo, si disponemos de clientes y direcciones, habiendo una tabla intermedia en la base de datos que permita relacionar uno con lo otro podríamos decir que hay una relación uno a muchos entre *Clientes* y *ClientesDirecciones*, así como también hay una relación uno a muchos entre *Direccion* y *ClientesDirecciones*. Desde el punto de vista de un modelador de entidades, una entidad intermedia no tiene ningún sentido. En realidad está ahí por el modo en el que se definen las estructuras relacionales, que como hemos comentado son un mundo diferente al que puede tener un programador orientado a objetos.

Con el fin facilitar la tarea a los modeladores de entidades, *EDM* cuando encuentra en el modelo relacional una relación de esta índole, **sin carga adicional**, automáticamente considera la misma como una asociación muchos a muchos.

Pongamos un ejemplo, partiremos del siguiente fragmento *SQL*, cuyo esquema puede verse en la figura 2.25.

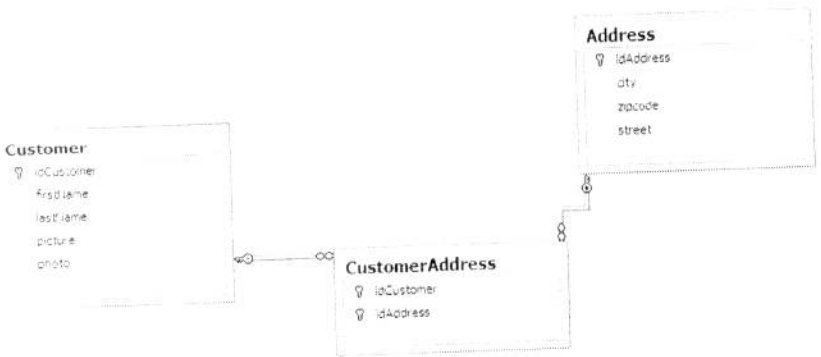


Figura 2.25.- Diagrama de una relación muchos a muchos en Sql Server

```
--SQL\EF4_EX_4_b.sql
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Customer](
    [idCustomer] [int] IDENTITY(1,1) NOT NULL,
    [firstName] [nvarchar](50) NOT NULL,
    [lastName] [nvarchar](50) NOT NULL,
    [picture] [image] NOT NULL,
    [photo] [image] NULL,
 CONSTRAINT [PK_Customre] PRIMARY KEY CLUSTERED
```

```
(
    [idCustomer] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Address](
    [idAddress] [int] IDENTITY(1,1) NOT NULL,
    [city] [nvarchar](50) NOT NULL,
    [zipcode] [nvarchar](50) NOT NULL,
    [street] [nvarchar](50) NOT NULL,
 CONSTRAINT [PK_Address] PRIMARY KEY CLUSTERED
(
    [idAddress] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[CustomerAddress](
    [idCustomer] [int] NOT NULL,
    [idAddress] [int] NOT NULL,
 CONSTRAINT [PK_CustomerAddress] PRIMARY KEY CLUSTERED
(
    [idCustomer] ASC,
    [idAddress] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

ALTER TABLE [dbo].[CustomerAddress]  WITH CHECK ADD  CONSTRAINT
[FK_CustomerAddress_Address] FOREIGN KEY([idAddress])
REFERENCES [dbo].[Address] ([idAddress])
GO
ALTER TABLE [dbo].[CustomerAddress] CHECK CONSTRAINT
[FK_CustomerAddress_Address]
GO

ALTER TABLE [dbo].[CustomerAddress]  WITH CHECK ADD  CONSTRAINT
[FK_CustomerAddress_Customer] FOREIGN KEY([idCustomer])
REFERENCES [dbo].[Customer] ([idCustomer])
GO
ALTER TABLE [dbo].[CustomerAddress] CHECK CONSTRAINT
[FK_CustomerAddress_Customer]
GO
```

Si creamos un proyecto de **Visual Studio 2010** con un modelo de entidades trabajando contra este esquema relacional, veremos cómo, automáticamente, *EDM* nos genera una asociación muchos a muchos entre las entidades en juego, Figura 2.26, *Customers* y *Address*.

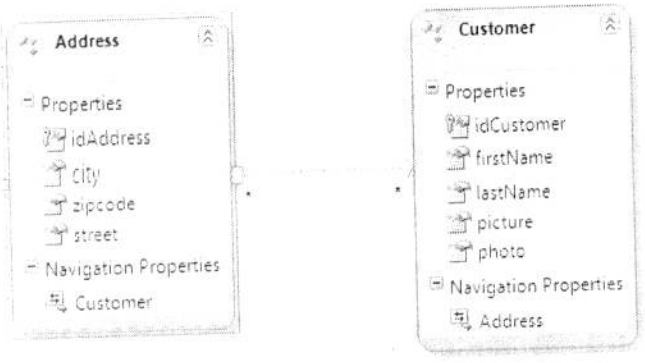


Figura 2.26.- Asociación muchos a muchos

Un elemento importante, resaltado en negrita unos párrafos más arriba, es que para que *EDM* cree de forma automática una relación de muchos a muchos la tabla intermedia de la base de datos no puede tener *carga adicional*. Con carga adicional normalmente se refiere a otras columnas que no sean las que relacionan las distintas tablas. Por ejemplo, entre *Pedidos* y *Productos* se podría pensar que hay una relación muchos a muchos, puesto que un *Producto* está en muchos *Pedidos* y un *Pedido* puede constar de muchos *Productos*. Sin embargo, si se modelara un escenario como este dentro de *EDM*, esta asociación no aparecería en el modelo como una asociación muchos a muchos. El motivo tiene que ver con el manejo de la carga adicional, es decir, en nuestro caso, con cómo se manejarían las columnas adicionales dentro de la entidad *LineaPedido* que relacionaría a las otras dos.

Para terminar con este punto referido a las asociaciones hablaremos un poco más detalladamente acerca de las restricciones en cascada, tocadas muy ligeramente en párrafos anteriores. Como usted bien sabrá, la mayoría de los motores relacionales ponen a nuestra disposición la posibilidad de usar distintas restricciones. Si por ejemplo entre las tablas *Customer* y *Order* de los ejemplos anteriores hubiéramos puesto una restricción de borrado en cascada dentro de la base de datos, *EDM* la habría incluido en su sección *SSDL*, es decir en el espacio de representación de lo físico. Esto nos permite delegar estas operaciones en cascada dentro del motor de la base de datos, algo en principio bueno ya que nos liberaría de cierto trabajo. Sin embargo, hay ocasiones en las que creamos un modelo de entidades a partir de una base de datos que no dispone de estas restricciones y las mismas podría resultar necesarias dentro de nuestro dominio.

Una posibilidad para resolver esto es mediante el establecimiento de las restricciones directamente en el modelo conceptual. En la ventana propiedades de la Figura 2.27 podemos ver como se ha establecido la restricción de cardinalidad *Cascade* para una asociación entre las entidades *Customer* y *Orders* vista anteriormente.

Nota: Esta técnica en concreto aplicada a la típica tabla de Clientes con una imagen es muy socorrida para evitar un posible problema de rendimiento que podría producirse si para traer la información de los clientes siempre tuviéramos que mover las fotografías de cada uno de ellos.

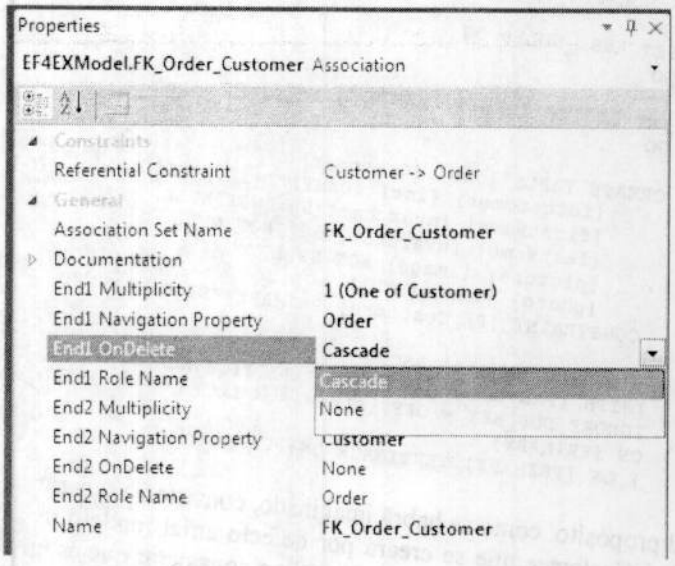


Figura 2.27.- Restricciones de cardinalidad

Lógicamente este cambio tiene su efecto en el espacio conceptual *CSDL*, y también tendrá su efecto a la hora de trabajar con este modelo de entidades, algo que deberemos de tener muy en cuenta puesto que el borrado en cascada implementado en el mundo conceptual solamente se realiza de aquellos elementos establecidos en la memoria, aunque este es un tema que por ahora no debemos tocar ya que no se ha hablado aún nada de cómo trabajar los modelos de entidades.

1.1.3.- Caso práctico: Table Splitting

Un ejemplo claro en el que la creación de un asociación nos aporta un gran valor dentro de la definición de un modelo de entidades es el uso de *Table Splitting*. Básicamente, es el hecho de dividir una entidad de nuestro modelo de entidades en varias, dos o más, y realizar asociaciones uno a uno entre la entidad principal y las nuevas entidades

creadas. Los razonamientos y elementos que motivan el uso de esta técnica son variados. Un motivo, por ejemplo, podría ser un deficiente diseño del modelo relacional por el cual una tabla tuviera una ingente cantidad de columnas y, por lo tanto, la entidad creada dentro de *EDM* dispusiera de una cantidad de propiedades equivalente. Otro de los ejemplos que nos podría hacer pensar en utilizar la técnica de *Table Splitting* sería la separación en un modelo de entidades de objetos valor, *Value Objects* y *Entities* por medio también de asociaciones uno a uno.

El caso práctico, que se propone a continuación, viene a enseñar esta técnica, para el caso de una entidad *Customer* que dispone de una propiedad *Photo* de tipo *image*, en *Sql Server*, **sql\ef4_ex5.sql**.

```
--SQL\EF4_EX_5.sql

SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[Customer](
    [idCustomer] [int] IDENTITY(1,1) NOT NULL,
    [firstName] [nvarchar](50) NOT NULL,
    [lastName] [nvarchar](50) NOT NULL,
    [picture] [image] NOT NULL,
    [photo] [image] NULL,
 CONSTRAINT [PK_Customre] PRIMARY KEY CLUSTERED
(
    [idCustomer] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
```

El propósito, como se habrá imaginado, consiste en separar la propiedad *Photo* de la entidad *Customer* que se creará por defecto en el modelo de entidades de *ADO.NET Entity Framework*. Puesto que es legítimo considerar que es un atributo que no siempre tendrá que formar parte de esta entidad y, por lo tanto, podría interpretarse como una navegación accesible solamente cuando se necesite.

Supongamos que ya hemos creado nuestro proyecto de *Visual Studio*, **code\ef4_ex_6**, y en él hemos incluido un modelo de entidades con la tabla mostrado en el fragmento *sql* anterior. La estructura de nuestra entidad, sería como la mostrada 2.28.

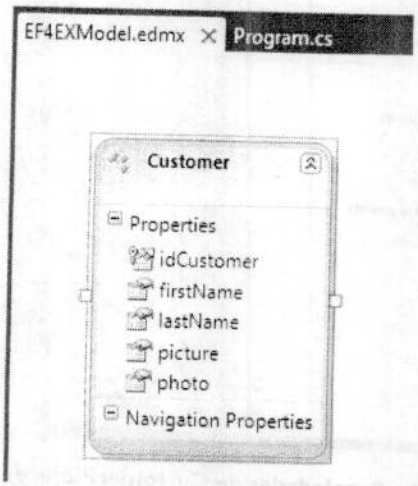


Figura 2.28.- Entidad Customer con la propiedad Photo

Si quisiéramos hacer nuestro *Table Splitting* con la propiedad *Photo* tendríamos que empezar por crear una nueva entidad dentro de *EDM*, en la que situar la propiedad *Photo*. Para ello, seleccionaremos la opción 'Agregar nueva entidad' desde el menú contextual de nuestro diseñador, figura 2.29. El asistente de creación de esta entidad nos permitirá seleccionar tanto el nombre como la propiedad clave de esta entidad. El resto de opciones se verán en este capítulo posteriormente. Para nuestro caso seleccionaremos como nombre de la entidad *CustomerPicture* con su correspondiente plural para el *EntitySet*, Figura 2.30

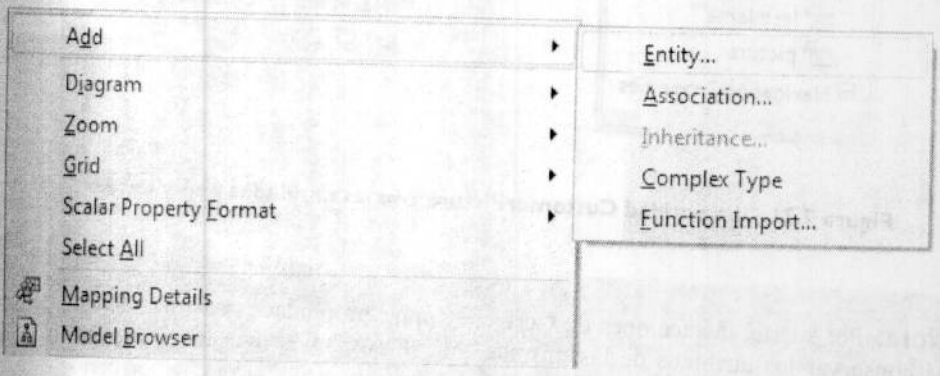


Figura 2.29.- Creación de nueva entidad



Figura 2.30.- Propiedades de CustomerPicture

Una vez que ya tenemos nuestra nueva entidad dentro del modelo de entidades podemos copiar las propiedades *idCustomer* y *photo* de la entidad *Customer* y pegarlas en la entidad *CustomerPicture*. Después podemos eliminar *photo* de la primera entidad.

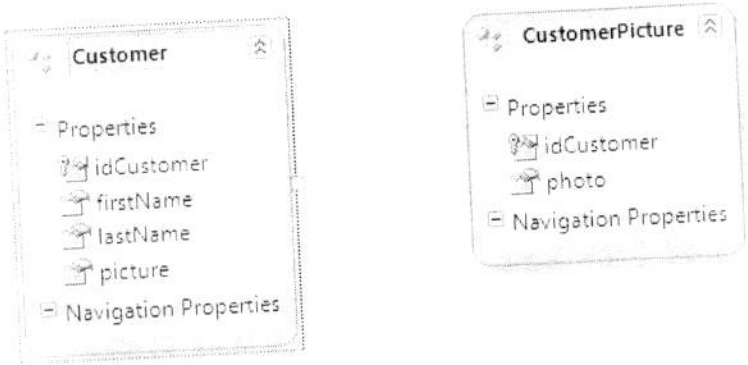


Figura 2.31.- La entidad CustomerPicture con las entidades pegadas

Nota: Por suerte, las acciones de Cortar y Copiar propiedades en un modelo EDM conservar los atributos de las mismas. Así, si copiamos una propiedad de una entidad a otra, los detalles como su tipo, máxima longitud, si puede ser o no nulo... se conservan.

Ahora solamente nos queda por indicar a *EDM* como hacemos corresponder esta entidad de nuestro modelo conceptual con el modelo físico subyacente, es decir, con el modelo relacional. Para ello mostramos la ventana de detalles de mapeo y hacemos que esta nueva entidad se *mapee* también con la entidad *Customer* como se puede ver a continuación.

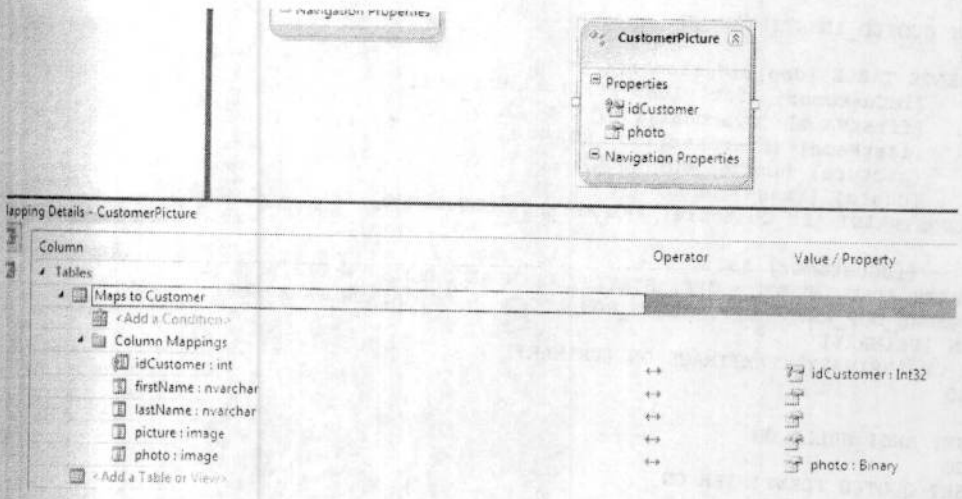


Figura 2.32.- Detalles de mapeo de CustomerPicture

Para finalizar nuestro ejemplo de *'Table Splitting'*, solamente nos queda realizar la asociación entre estas dos entidades. Para ello, aunque no exista integridad referencial como tal haremos doble clic en el elemento asociación y estableceremos la misma como sigue, Figura 2.33.

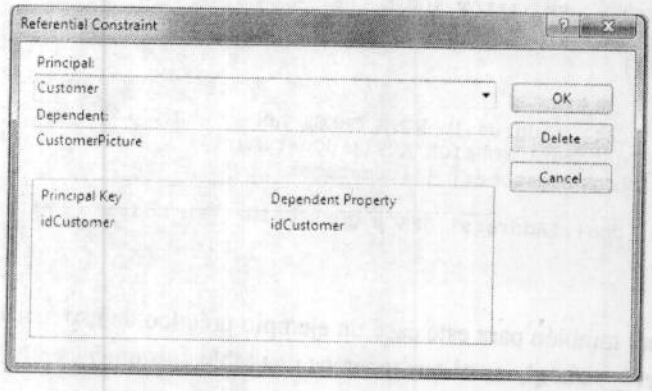


Figura 2.33.- Detalles de la asociación

1.1.4.- Caso práctico: Entity Splitting

Entity Splitting se podría definir como el caso contrario a *Table Splitting*, si en el caso anterior una entidad se separaba en distintas entidades con asociaciones entre cada una de ellas, *Entity Splitting* nos permitirá juntar en una misma entidad varias tablas definidas en el modelo relacional subyacente.

```
--SQL\EF4_EX_6.sql

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Customer](
    [idCustomer] [int] IDENTITY(1,1) NOT NULL,
    [firstName] [nvarchar](50) NOT NULL,
    [lastName] [nvarchar](50) NOT NULL,
    [picture] [image] NOT NULL,
    [photo] [image] NULL,
 CONSTRAINT [PK_Customre] PRIMARY KEY CLUSTERED
(
    [idCustomer] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Address](
    [idCustomer] [int] NOT NULL,
    [street] [nvarchar](50) NOT NULL,
    [city] [nvarchar](50) NOT NULL,
    [zipcode] [nvarchar](10) NOT NULL,
 CONSTRAINT [PK_Address_1] PRIMARY KEY CLUSTERED
(
    [idCustomer] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

ALTER TABLE [dbo].[Address]  WITH CHECK ADD  CONSTRAINT
[FK_Address_Customer] FOREIGN KEY([idCustomer])
REFERENCES [dbo].[Customer] ([idCustomer])
GO
ALTER TABLE [dbo].[Address] CHECK CONSTRAINT [FK_Address_Customer]
GO
```

Con el fin de poner también para este caso un ejemplo práctico de uso partiremos de un esquema, **sql\ef4_ex6.sql**, en el que tenemos una tabla *Customers* y una tabla

Address con una relación 1..0-1. El objetivo, una vez creado el modelo de entidades, será juntar los datos de ambas tablas dentro de una misma entidad. Si partimos de una solución con el modelo de entidades creado por defecto por *ADO.NET EF 4.1* para el relacional anterior, Figura 2.34, **code\ef4_ex7,** solamente tendremos que hacer el trabajo de juntar nuestras tablas en una sola entidad.

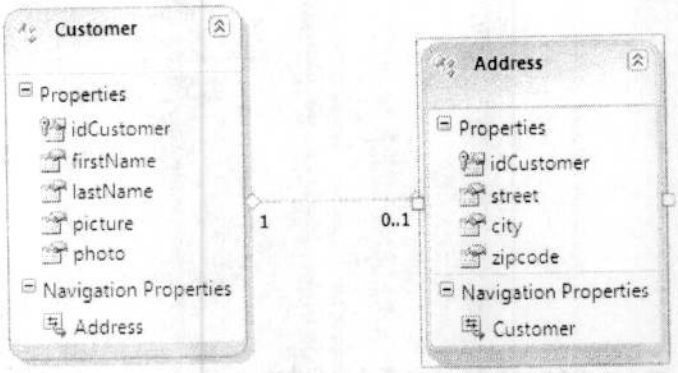


Figura 2.34.- Entity Splitting

Como es de suponer, el primer paso es hacer que la entidad *Customer* disponga de las propiedades de la entidad *Address*, excepto lógicamente la propiedad *idCustomer*. Para ello, lo único que tendremos que hacer en el diseñador es seleccionarlas de la entidad *Address* y cortarlas, para posteriormente pegarlas en la entidad destino *Customer*, ver figura 2.35. Una vez hecho esto, ya podemos eliminar la entidad Address y por lo tanto su asociación.

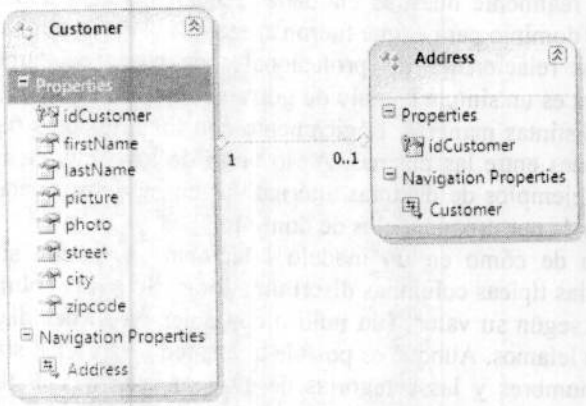


Figura 2.35.- Pegando las entidades de Address

Una vez que ya hemos terminado de diseñar nuestra entidad solamente nos quedará modificar los detalles de mapeo, es decir, cómo juntamos nuestros espacios **CSDL** y **SSDL**. Para ello, en las opciones de detalle de mapeo de nuestra entidad *Customer* agregaremos una nueva tabla de mapeo, o sea, la misma entidad a dos tablas, tal cual se muestra en la figura 2.36.

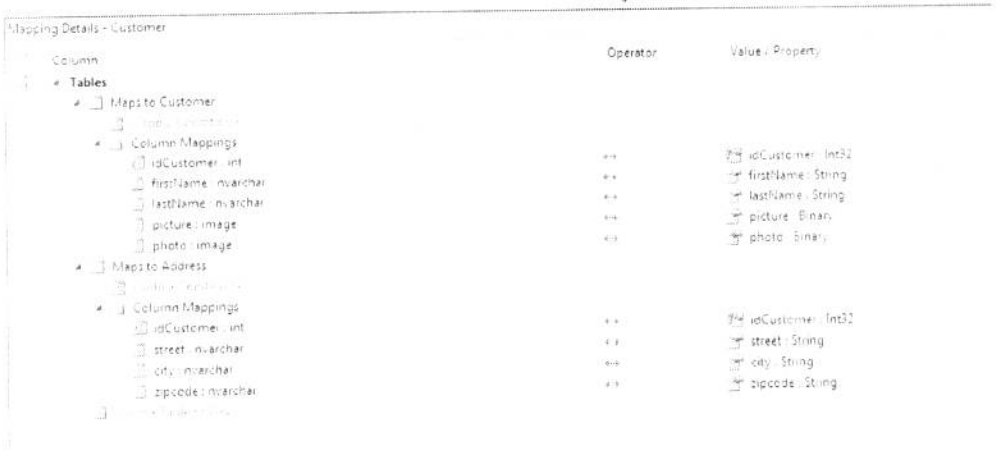


Figura 2.36.- Mapeo con Table Splitting

1.1.5.- Caso práctico: La herencia

La herencia es, sin lugar a dudas, una de las características más importantes que tenemos a la hora de hacer modelos de dominio. Es además uno de los elementos que a simple vista mejor nos permiten diferenciarnos de un modelo relacional.

Un desarrollador orientado a objetos está acostumbrado tanto a la herencia como al polimorfismo y, por lo tanto, soportar esta característica en modelos de dominio es ofrecer una vía para que realmente nuestras entidades tengan más capacidades para resolver los problemas del dominio para el que fueron creadas.

Dentro de los modelos relacionales los profesionales de bases de datos (si, ya sabemos que muchas veces es un simple cambio de gorra de un desarrollador) intentan 'simular' la herencia de distintas maneras. Lógicamente con los artefactos de los que disponen: tablas y relaciones entre las mismas. A lo largo de los siguientes párrafos intentaremos ir poniendo ejemplos de distintas alternativas en modelos relacionales y cómo interpretarlas dentro de nuestros modelos de dominio.

El ejemplo más claro de cómo en un modelo relacional se intenta simular la herencia es por medio de las típicas columnas discriminadoras. Son esas columnas que todos hemos creado que, según su valor (un nulo o cualquier otro), decidíamos que otras columnas de la tabla leíamos. Aunque es posible que usted no lo sepa, suele pasar mucho además con los nombres y las categorías de algunos patrones. Este tipo de *artimaña* recibe el nombre de "**Herencia por Jerarquia**". En la literatura especializada se le suele llamar en realidad **TPH**, siglas de su nombre en lengua inglesa *Table Per*

Hierarchy. Tal y como hemos hecho hasta ahora pondremos un ejemplo, algo que nos suele ayudar a los programadores de una forma mejor que unas cuantas páginas de texto. Para este ejemplo partiremos de la siguiente definición de tabla:

```
--SQL\EF4_EX_7.sql

GO


SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[Audio](
    [idAudio] [int] IDENTITY(1,1) NOT NULL,
    [code] [nvarchar](200) NULL,
    [songLetter] [nvarchar](max) NULL,
    [songDuration] [time](7) NULL,
    [podcastUri] [nvarchar](50) NULL,
    [podcastFormat] [nvarchar](10) NULL,
 CONSTRAINT [PK_Audio] PRIMARY KEY CLUSTERED
(
    [idAudio] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]

GO
```

Aunque es un ejemplo un poco forzado, se puede observar de un simple vistazo como en esta tabla se dispone de dos elementos, *podcast* y *song*, que conceptualmente deberían estar separados. Para realizar esta *separación*, en el modelo relacional, se ha situado una nueva columna, *code*, cuyo valor solamente tiene sentido operacionalmente para comprobar si los datos de esa fila se refieren a *song* o *podcast*. Lógicamente, esto no es más que un pequeño "apaño" que podemos hacer en un modelo relacional al carecer de algo tan común para un programador orientado a objetos como la herencia.

Tal y como hemos comentado en el primer capítulo, aunque fuera un poco por encima, cuando uno se decide a crear un modelo de dominio debería realmente pensar en los conceptos del mismo y las entidades que lo forman, de manera independiente a como sea o vaya a ser la representación del modelo relacional.

Entity Data Model pondrá a nuestra disposición la posibilidad de modelar esta tabla en una jerarquía de entidades como la que seguramente ya hemos pensado: una entidad *Audio* y dos subclases de esta, *Song* y *Podcast*.

Con el fin de enseñar esta característica, crearemos un nuevo proyecto de Visual Studio y agregaremos un modelo de entidades de ADO.NET referenciando a la base de datos que contenga la tabla mostrada en el script anterior, **code\ef4_ex_8**. Una vez creado este modelo de entidades, la entidad que por defecto incluye EDM es similar a la que podemos ver en la siguiente figura, figura 2.37.



Figura 2.37.- Model de entidades con Audio

Lógicamente, si queremos conseguir nuestro propósito, a este modelo deberemos agregarle nuevas entidades tanto para *Song* como para *Podcast*. Lo podemos conseguir por distintos caminos. Bien seleccionando desde nuestra ventana de herramientas de Visual Studio el elemento *Entity*, Figura 2.38, o bien desde el menú contextual de la superficie de diseño de Entity Data model, figura 2.39.

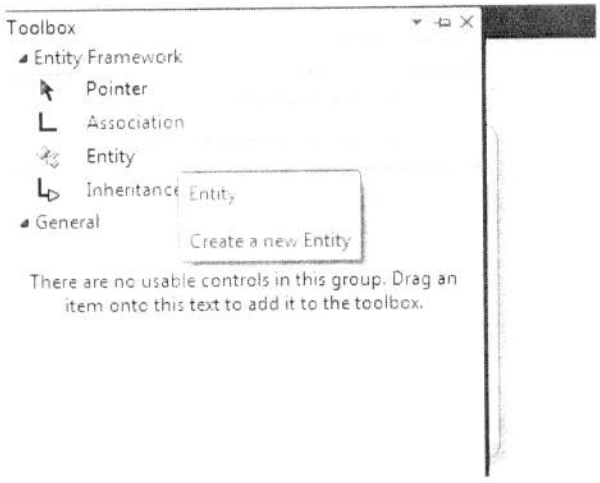


Figura 2.38.- Agregar entidad desde la ventana de herramientas

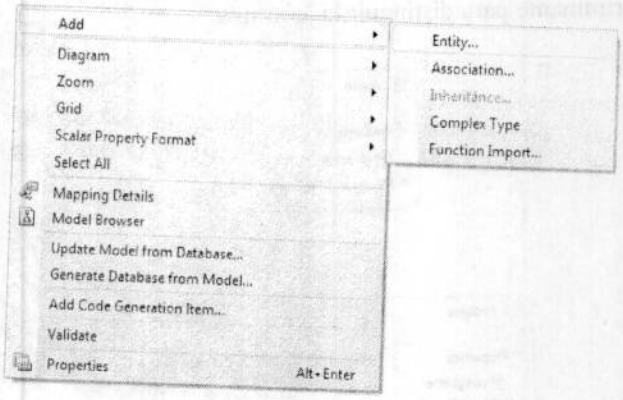


Figura 2.39.- Agregando una entidad desde la superficie de diseño

El asistente de creación de una entidad nos permitirá seleccionar desde el nombre del *EntityType* y *EntitySet* hasta la posibilidad de seleccionar o no una clave primaria. Nosotros para nuestro ejemplo solamente estableceremos el nombre, sin establecer clase base o propiedad clave, como se puede ver en la figura 2.40.

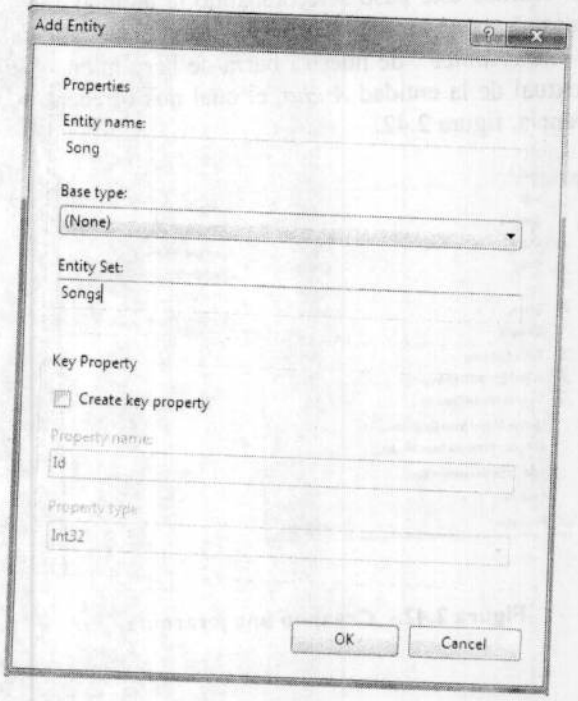


Figura 2.40.- Asistente de creación de una entidad

Una vez creadas las entidades, desde el propio diseñador podemos copiar/cortar las propiedades de la entidad *Audio* que queramos mover hasta nuestras nuevas entidades, con el fin de que las mismas queden tal y como se muestran en la figura 2.41. Fíjese que la propiedad *code* ha desaparecido, la hemos eliminado, porque en realidad, conceptualmente no aporta nada, aunque dentro de pocas líneas veremos cómo hacer uso de este discriminante para distinguir la jerarquía.



Figura 2.41.- Entidades Song y Podcast con sus propiedades

Una vez dispuestas las entidades en nuestro diseñador, lógicamente nos quedarán por efectuar los pasos necesarios para crear la jerarquía. Como se comentó en una nota anterior, podríamos ahorrarnos este paso seleccionando la entidad base en el asistente de creación de entidades. Aunque también es posible llevarlo a cabo seleccionando el elemento *"Herencia – Inheritance"* de nuestra barra de herramientas, lo haremos por medio del menú contextual de la entidad *Audio*, el cual nos ofrecerá la posibilidad de agregar una nueva herencia, figura 2.42.

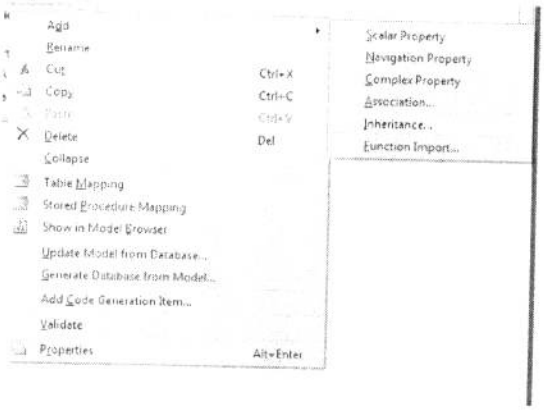


Figura 2.42.- Creando una jerarquía

Una vez terminado el proceso de creación de las jerarquías, Figura 2.43, ya tendremos nuestro modelo de entidades como pensamos en un principio. Lógicamente, falta todo el trabajo de *mapear* estas entidades con el modelo relacional subyacente, y es aquí, dónde entra el trabajo de nuestra columna *code*.

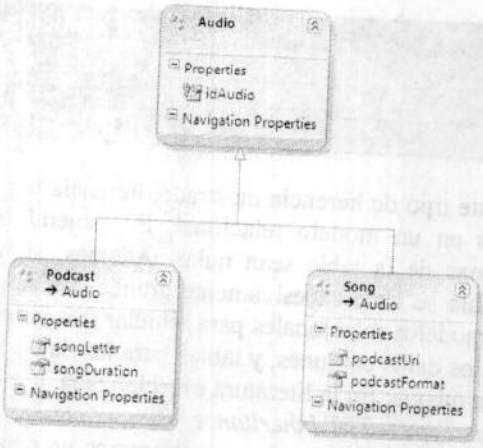


Figura 2.43.- Jerarquía terminada

Para cada una de las nuevas entidades tendremos que acudir a sus detalles de mapeo y seleccionar con quién y cómo se mapea. Como ya sabemos a estas alturas, la ventana de *Mapping Details* nos permitirá realizar esta tarea. Para nuestro caso tanto *Song* como *Podcast* se mapearán con la entidad *Audio*. Pero entonces ¿cómo sabrá EF para cada fila a que entidad hacerla corresponder?. La respuesta es que lo hará por medio de la columna discriminadora. Para hacer uso de la misma en la ventana de "Detalles de mapeo", solamente tenemos que incluir una nueva condición. Por ejemplo para nuestro caso que el valor de la columna *code* sea igual a *podcast* o *song*, Figura 2.44.

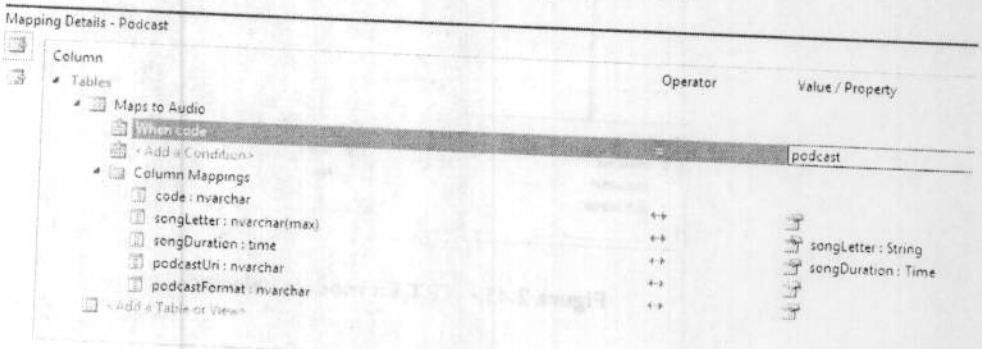


Figura 2.44.- Estableciendo una condición

Lógicamente, deberemos tener completitud en las condiciones. Es decir, que si un elemento de la jerarquía posee una condición en su mapeo (por ejemplo que el valor de la columna sea un nulo), debería existir otro elemento en la jerarquía con la condición de no nulo para la columna discriminadora seleccionada. En nuestro caso las condiciones que tendremos en los mapeos de las entidades serán las descritas en la siguiente tabla. También puede revisarlas de una forma más gráfica y seguramente más cómoda en el correspondiente ejemplo de código **code\ef4_ex_8.**

Tabla 2.- Condiciones establecidas en cada entidad

Entidad	Condición
Audio	Code IsNull
Song	Code = song
Podcast	Code = podcast

El uso de este tipo de herencia mostrado, herencia por jerarquía, no siempre es una buena solución en un modelo relacional. En general impone la necesidad de que muchas columnas de la tabla sean nulas. Además, si hay muchos elementos en la jerarquía, la tabla se hace excesivamente grande con mucho tamaño por página. Una alternativa en modelos relacionales para simular una jerarquía consiste en disponer de una tabla para los datos comunes, y tablas para los datos específicos de cada uno de los miembros de la misma. En la literatura especializada, a este tipo de técnica se la conoce como *Table Per Type – TPT Inheritance.*

Al igual que hemos hecho antes, mostraremos un ejemplo, **code\EF4_EX_9**, paso a paso e iremos viendo en EDM la construcción del modelo de entidades con herencia a partir de un esquema relacional con TPT. Imaginemos que partimos del siguiente esquema relacional, el mostrado en la Figura 2.45, y que tiene por DDL lo mostrado en el listado a continuación.

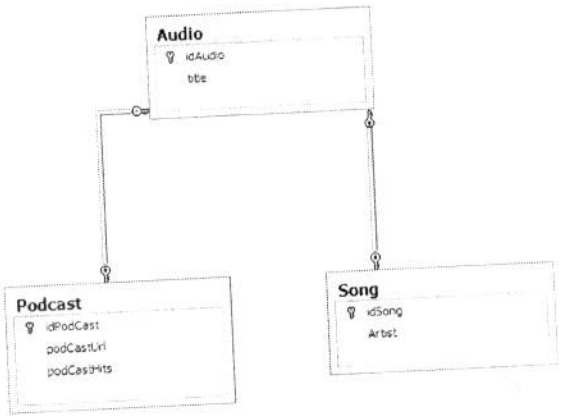


Figura 2.45.- TPT En modelo relacional

```
--SQL\EF4_EX_8.sql
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Audio](
    [idAudio] [int] IDENTITY(1,1) NOT NULL,
    [title] [nvarchar](200) NOT NULL,
 CONSTRAINT [PK_Audio] PRIMARY KEY CLUSTERED
(
    [idAudio] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Song](
    [idSong] [int] NOT NULL,
    [Artist] [nvarchar](200) NOT NULL,
 CONSTRAINT [PK_Song_1] PRIMARY KEY CLUSTERED
(
    [idSong] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Podcast](
    [idPodCast] [int] NOT NULL,
    [podCastUrl] [nvarchar](50) NOT NULL,
    [podCastHits] [int] NOT NULL,
 CONSTRAINT [PK_Podcast] PRIMARY KEY CLUSTERED
(
    [idPodCast] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

ALTER TABLE [dbo].[Song]  WITH CHECK ADD  CONSTRAINT [FK_Song_Audio]
FOREIGN KEY([idSong])
REFERENCES [dbo].[Audio] ([idAudio])
GO
ALTER TABLE [dbo].[Song] CHECK CONSTRAINT [FK_Song_Audio]
GO

ALTER TABLE [dbo].[Podcast]  WITH CHECK ADD  CONSTRAINT
[FK_Podcast_Audio] FOREIGN KEY([idPodCast])
REFERENCES [dbo].[Audio] ([idAudio])
```

```
GO
ALTER TABLE [dbo].[Podcast] CHECK CONSTRAINT [FK_Podcast_Audio]
GO
```

Como se puede apreciar en el esquema y el fragmento de código anterior, la simulación de la herencia en el modelo relacional se realiza mediante una tabla común cuyo propósito es el de mantener los datos comunes a todos los elementos, en este caso *Song* y *PodCast*, y una tabla para cada uno de los elementos pertenecientes a la jerarquía.

Si incluimos las tres tablas anteriores, *Audio*, *Song* y *PodCast*, en un modelo de EDM, el diseñador, por defecto, nos mostrará la misma estructura de la que disponemos en el modelo relacional, es decir, tres entidades con asociaciones 1..0,1 entre ellas, Figura 2.46. Nuestro trabajo consistirá en ver cómo modificamos este modelo por uno con herencia entre las entidades anteriores, teniendo como base la entidad *Audio*.

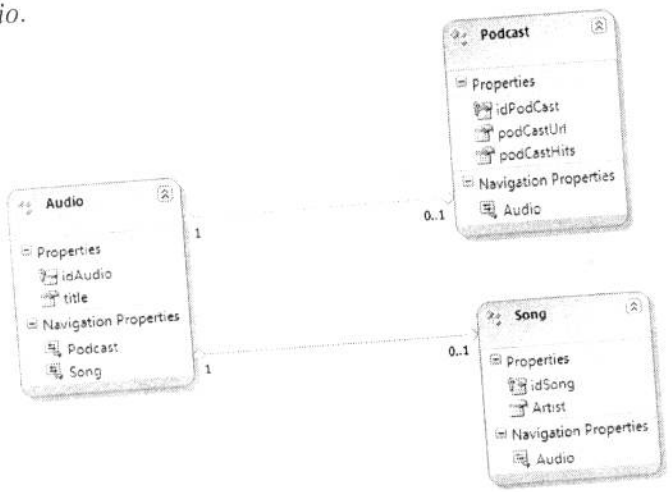


Figura 2.46.- Modelo de EDM antes de realizar la herencia

Para realizar nuestro propósito el primer paso será el de eliminar las asociaciones existentes entre las entidades, con lo cual, las propiedades de navegación se eliminan del modelo. Una vez eliminadas las asociaciones, al igual que en el ejemplo anterior, incluiremos una nueva herencia desde la ventana de herramientas o directamente en el menú contextual de la entidad o modelo, entre *Audio* y *Song*, por poner un ejemplo.

Una vez agregada esta herencia, tendremos que modificar el *mapeo* de esta entidad con el modelo relacional subyacente, algo que ya sabemos hacer desde nuestra ventana de *"Detalles de mapeo"*. El **truco** en este punto, no es más que el de asignar correctamente el mapeo de las claves de cada subclase a la clave primaria de la clase base, en el caso que nos ocupa la asignación de *idPodCast a idAudio*, como se puede ver en la siguiente figura.

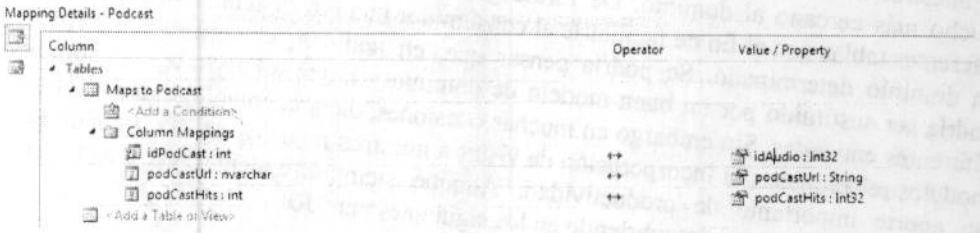


Figura 2.47.- Detalles de mapeo de Audio y Song

Lógicamente, los valores de *idPodCast* e *idSong* no tendrán más sentido que el valor dado en *idAudio*, al que están mapeados, por lo que estas propiedades de la entidad tendrán que ser eliminadas. El resultado de todo este trabajo realizado puede verse en la Figura 2.48.

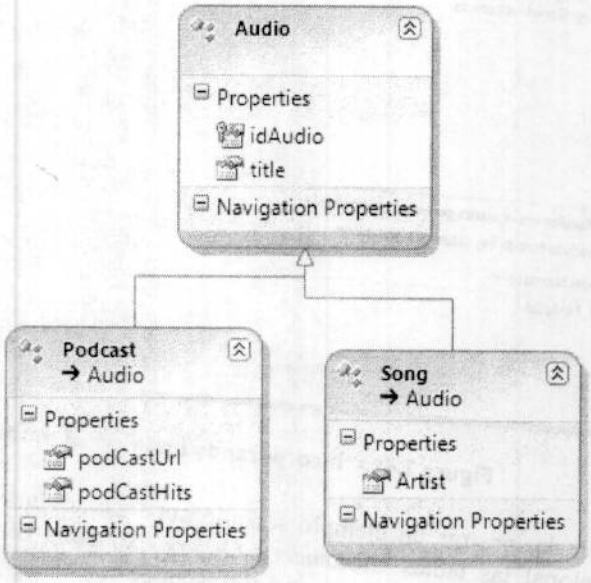


Figura 2.48.- Ejemplo TPT terminado

1.1.6.- Caso práctico: Defining Query

Hasta ahora, en lo que respecta a nuestros modelos relacionales, solamente hemos visto como incorporar elementos básicos como tablas y relaciones entre las mismas de nuestra base de datos. Por supuesto, además de tablas, en las bases de datos existen otros elementos que podrían ser interesantes para incorporar a nuestros modelos de dominio, como por ejemplo las vistas y/o los procedimientos almacenados.

En este caso práctico nos centraremos en las vistas y en cómo darle soporte dentro de nuestros modelos de EDM. Una vista en un modelo relacional en realidad es algo mucho más cercano al dominio. De forma general se utilizan para agregar datos de diferentes tablas con el fin de facilitarle al consumidor una información más adaptada a un dominio determinado. Se podría pensar que, en realidad, el concepto de vistas podría ser sustituido por un buen modelo de dominio y la consulta y/o agregación de diferentes entidades. Sin embargo en muchas ocasiones, dada la complejidad de ciertos modelos relacionales, la incorporación de vistas a nuestros modelos de EDM puede ser un aporte importante de productividad. Aunque siempre teniendo en cuenta las limitaciones que iremos descubriendo en los siguientes párrafos.

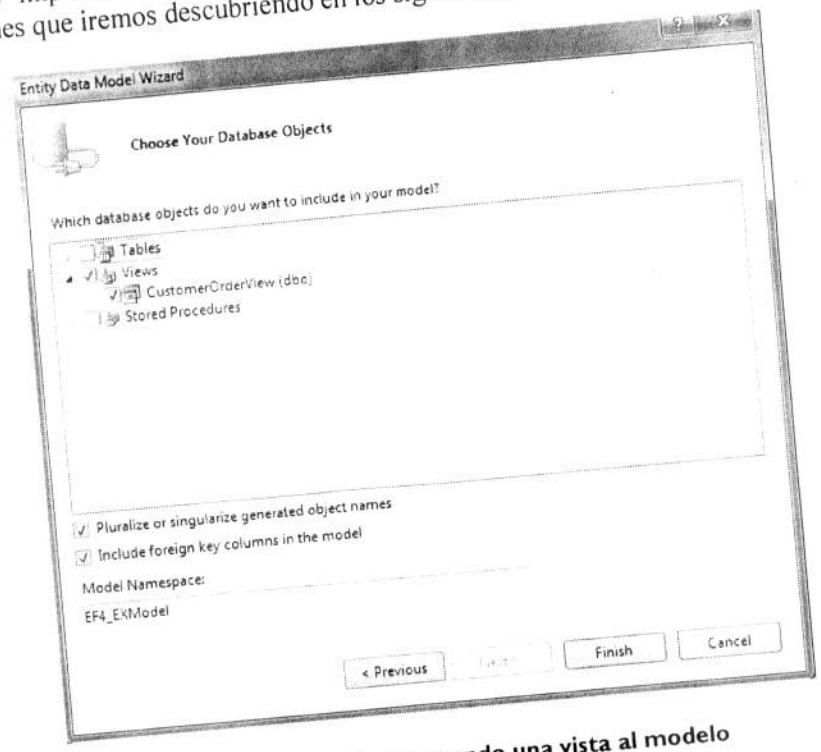


Figura 2.49.- Incorporando una vista al modelo

Con el fin de mostrar un ejemplo sencillo supongamos que tenemos en nuestro modelo relacional las tablas *Customer* y *Orders* expuestas con anterioridad, y que sobre estas tablas se crea una vista que se corresponde con el siguiente fragmento:

```
--SQL\EF4_EX_9.sql

SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE VIEW [dbo].[CustomerOrderView]
```

```
AS
SELECT     dbo.[Order].totalOrder, dbo.[Order].orderDate,
dbo.Customer.idCustomer
FROM         dbo.Customer INNER JOIN
                      dbo.[Order] ON dbo.Customer.idCustomer =
dbo.[Order].idCustomer

GO
```

Ahora, si creamos un nuevo proyecto con un modelo de entidades, **code\EF4_EX_10**, y en él incorporamos nuestra nueva vista por medio del asistente, figura 2.49, podremos ver con EDM nos muestra una 'entidad' con algunas peculiaridades.

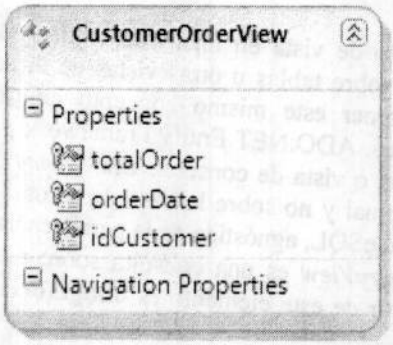


Figura 2.50.- Entidad creada a partir de una vista

Lo primero que observamos en esta entidad creada a partir de una vista, es que contiene en todas sus propiedades el atributo de clave. Este hecho se debe a que EDM no encuentra una definición de clave, algo lógico por otra parte, y ante esto establece en todas las propiedades. Si lo desea, puede ir eliminando uno a uno estos atributos, aunque al menos una propiedad debe de conservarlo.

Otro elemento importante a conocer con el trabajo de las vistas es que la definición en el espacio **SSDL** de EDM es muy distinto de la definición de una tabla. Si abrimos el archivo **edmx** anterior con un editor XML (puede seleccionar en el menú contextual de Visual Studio *Abrir Con*) veremos como la definición del espacio SSDL contiene algo similar a:

```
<EntitySet Name="CustomerOrderView"
EntityType="EF4_EXModel.Store.CustomerOrderView" store:Type="Views"
store:Schema="dbo" store:Name="CustomerOrderView">
            <DefiningQuery>SELECT
      [CustomerOrderView].[totalOrder] AS [totalOrder],
      [CustomerOrderView].[orderDate] AS [orderDate],
      [CustomerOrderView].[idCustomer] AS [idCustomer]
      FROM [dbo].[CustomerOrderView] AS
```

```
[CustomerOrderView]</DefiningQuery>
</EntitySet>
```

En esta definición de nuestro espacio SSDL, se puede observar cómo se define un elemento de nuestro almacén como **CustomerOrderView**, y como el mismo se corresponde con una consulta en el dialecto específico de la base de datos.

Llegados hasta aquí, realmente podría pensar que en verdad **EDM** no necesita conocer nada acerca de la definición de la vista, es decir, en realidad, dentro de nuestro elemento **DefiningQuery** podríamos poner cualquier tipo de consulta, por ejemplo la propia definición de la vista.

Lógicamente, las consultas de definición, pueden suponer un punto crítico en el posible uso de EDM en aplicaciones multi-base de datos, puesto que, cuantos más elementos de tipo **DefiningQuery** tengamos en nuestro SSDL, sean vistas o consultas expresadas directamente, más elementos tendremos que adaptar a los distintos motores.

1.1.7.- Caso práctico: Entidades de solo lectura

Al igual que el concepto de vista en un modelo relacional nos permite definir una consulta de agregación sobre tablas u otras vistas de la base de datos, en ocasiones, puede ser interesante tener este mismo concepto dentro de nuestros modelos de dominio. Para estos casos, ADO.NET Entity Framework pone a nuestra disposición el concepto de **QueryView** o vista de consulta. Una *QueryView* representa una consulta sobre el espacio conceptual y no sobre la base de datos física, por lo tanto, la misma tiene que estar escrita en eSQL, agnóstico de la persistencia subyacente.

Puesto que una *QueryView* es una consulta sobre nuestro modelo conceptual, la entidad generada a partir de este elemento es **obligatoriamente una entidad de solo lectura**.

QueryView nos permite habilitar los siguientes escenarios dentro de nuestros modelos conceptuales:

- Definir entidades en nuestro modelo conceptual que no incluyan todas las propiedades de una entidad dada y hacerla de solo lectura.
- Mapear columnas calculadas a una entidad a partir de una consulta del modelo conceptual.
- Definir funciones de partición para las entidades, posibilitando crear entidades de cada una de las particiones que definamos.
- Mapear múltiples tipos a una única tabla.
- Usar filtros condicionales para definir entidades conceptuales.

Por desgracia, el uso de *QueryView* también acarrea ciertos condicionantes que es necesarios conocer y valorar:

- Estas entidades son de solo lectura: los cambios solamente se podrán realizar mediante el uso de funciones de actualización.

- Cuando se define una entidad mediante una *QueryView*, todas las entidades que se relacionan con ella también deberán estar definidas por medio de elementos de tipo *QueryView*.

Volvemos en este caso a realizar un ejemplo que nos permita entender mejor el funcionamiento y la construcción de una *QueryView*. Para ello haremos algo similar al punto anterior, partiendo del siguiente modelo relacional:

```
--SQL\EF4_EX_10.sql

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Customer](
    [idCustomer] [int] IDENTITY(1,1) NOT NULL,
    [firstName] [nvarchar](200) NOT NULL,
    [lastName] [nvarchar](50) NOT NULL,
    [picture] [image] NOT NULL,
    [photo] [image] NULL,
    [passportNumber] [nvarchar](15) NOT NULL,
 CONSTRAINT [PK_Customre] PRIMARY KEY CLUSTERED
(
    [idCustomer] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Order](
    [idOrder] [int] IDENTITY(1,1) NOT NULL,
    [orderDate] [date] NOT NULL,
    [totalOrder] [numeric](12, 2) NOT NULL,
    [idCustomer] [int] NOT NULL,
 CONSTRAINT [PK_Order] PRIMARY KEY CLUSTERED
(
    [idOrder] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

ALTER TABLE [dbo].[Order]  WITH CHECK ADD  CONSTRAINT
[FK_Order_Customer] FOREIGN KEY([idCustomer])
REFERENCES [dbo].[Customer] ([idCustomer])
ON DELETE CASCADE
GO
ALTER TABLE [dbo].[Order] CHECK CONSTRAINT [FK_Order_Customer]
GO
```

Crearemos, al igual que hasta ahora, un proyecto de Visual Studio en el que incorporaremos un modelo de entidades con las tablas anteriores, **code\EF4_EX_11**.

Una vez creado el modelo agregaremos una nueva entidad, desde la barra de herramientas o bien desde el menú contextual del diseñador, en la que incluiremos las propiedades que nos interesen, en nuestro ejemplo las mostradas en la figura 2.51.



Figura 2.51.-Creación de una entidad de solo lectura

Una vez creada esta entidad tendremos que especificar su mapeo. Lógicamente no podemos basarnos en nuestra ventana de *detalles de mapeo* puesto que ésta solamente nos permite realizar mapeos de entidades con elementos del almacén subyacente, y ahora a nosotros nos interesa realizar el mapeo contra una consulta de nuestro modelo conceptual. Por desgracia, aún en esta segunda versión, este trabajo sigue sin estar soportado en el diseñador directamente. Por lo tanto abriremos nuestro fichero *edmx* con un editor de xml, por ejemplo mediante la opción *abrir con* del menú contextual del archivo en el *explorador de soluciones*. Una vez abierto el fichero, buscaremos la sección M-S la cual, por defecto, tendrá el siguiente aspecto.

```
<edmx:Mappings>
      <Mapping Space="C-S"
xmlns="http://schemas.microsoft.com/ado/2008/09/mapping/cs">
        <EntityContainerMapping
StorageEntityContainer="EF4_EXModelStoreContainer"
CdmEntityContainer="EF4_EXEntities">
          <EntitySetMapping Name="Customers">
            <EntityTypeMapping TypeName="EF4_EXModel.Customer">
              <MappingFragment StoreEntitySet="Customer">
                <ScalarProperty Name="idCustomer"
ColumnName="idCustomer" />
                <ScalarProperty Name="firstName"
ColumnName="firstName" />
                <ScalarProperty Name="lastName" ColumnName="lastName"
/>
                <ScalarProperty Name="picture" ColumnName="picture"
/>
                <ScalarProperty Name="photo" ColumnName="photo" />
                <ScalarProperty Name="passportNumber"
ColumnName="passportNumber" />
              </MappingFragment>
            </EntityTypeMapping>
```

```
            </EntitySetMapping>
            <EntitySetMapping Name="Orders">
              <EntityTypeMapping TypeName="EF4_EXModel.Order">
                <MappingFragment StoreEntitySet="Order">
                  <ScalarProperty Name="idOrder" ColumnName="idOrder"
/>
                  <ScalarProperty Name="orderDate"
ColumnName="orderDate" />
                  <ScalarProperty Name="totalOrder"
ColumnName="totalOrder" />
                  <ScalarProperty Name="idCustomer"
ColumnName="idCustomer" />
                </MappingFragment>
              </EntityTypeMapping>
            </EntitySetMapping>
          </EntityContainerMapping>
        </Mapping>
</edmx:Mappings>
```

Como podrá observar, dentro de esta sección se definen los mapeos para las entidades *Customer* y *Order*. Para nuestra nueva entidad *CustomerQueryView* incluiremos un nuevo elemento dentro de *EntitySetMapping* con lo siguiente:

```
<EntitySetMapping Name="CustomerQueryViews">
            <QueryView>
              SELECT VALUE
EF4_EXModel.CustomerQueryView(c.idCustomer,o.idOrder,c.firstName,o.to
talOrder)
              FROM EF4_ExModelStoreContainer.Customer as c LEFT JOIN
EF4_ExModelStoreContainer.Order as o ON c.idCustomer = o.idCustomer
            </QueryView>
</EntitySetMapping>
```

El fragmento anterior, por medio del elemento **QueryView** nos permitirá escribir nuestra consulta directamente en **eSQL** dejando claro por medio del constructor *SELECT VALUE EF4_EXModel.CustomerQueryView* cómo se construyen cada una de las entidades. En realidad, esto es similar a una vista con un *DefiningQuery*, con la importante diferencia que mientras una *DefiningQuery* se escribe en un dialecto concreto de una base de datos, una *QueryView* se escribe en *Entity SQL*.

1.1.8.- Caso práctico: Procedimientos almacenados

El soporte para los procedimientos almacenados es otro de los puntos dónde el equipo de producto ha trabajado más en esta segunda versión. Si bien puede que esto no le parezca interesante, puesto que el trabajo con procedimientos almacenados choca frontalmente con la definición de modelos de dominio y el principio de la ignorancia de la persistencia, también es posible que se convierta en una pieza destacable de su trabajo con EF al permitir dar soporte a modelos *legacy* (sistemas antiguos) dentro de sus nuevos productos.

Empezaremos viendo las capacidades de trabajo con procedimientos almacenados por medio de su uso en los métodos CRUD de las entidades (*Create – Update –*

Delete), para ello, partiremos de una tabla Customer y los procedimientos almacenados que se definen a continuación:

```
--SQL\EF4_EX_11.sql

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Customer](
    [idCustomer] [int] IDENTITY(1,1) NOT NULL,
    [firstName] [nvarchar](200) NOT NULL,
    [lastName] [nvarchar](50) NOT NULL,
    [picture] [image] NULL,
    [passportNumber] [nvarchar](15) NOT NULL,
 CONSTRAINT [PK_Customre] PRIMARY KEY CLUSTERED
(
    [idCustomer] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[UpdateCustomer]

    @FName nvarchar(200),
    @LName nvarchar(200),
    @PNumber nvarchar(5),
    @ID integer
AS
BEGIN

    SET NOCOUNT ON;

    UPDATE dbo.Customer SET firstName = @FName,lastName
=@LName,passportNumber=@PNumber
    WHERE idCustomer = @ID

END
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[DeleteCustomer]

    @ID integer
AS
BEGIN

    SET NOCOUNT ON;

    DELETE dbo.Customer
    WHERE idCustomer = @ID
```

```
END
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[CreateCustomer]

    @FName nvarchar(200),
    @LName nvarchar(200),
    @PNumber nvarchar(5)
AS
BEGIN

    SET NOCOUNT ON;

    INSERT INTO dbo.Customer VALUES(@FName,@LName,NULL,@PNumber)
END
GO
```

Creamos un nuevo proyecto, **code\EF4_EX_12,** e incorporamos nuestra tabla *Customer* y los tres procedimientos almacenados anteriores en un nuevo modelo de EDM. Ahora, una vez que ya dispongamos de la entidad, seleccionaremos su ventana de detalles de mapeo. En ella podremos ver una nueva opción desconocida hasta el momento. Si se fija, en los botones situados a la izquierda de la ventana en orientación vertical, disponemos de la opción de especificar, a mayores del mapeo propiedades columnas, el mapeo de los métodos CRUD con procedimientos almacenados, figuras 2.52 y 2.53.

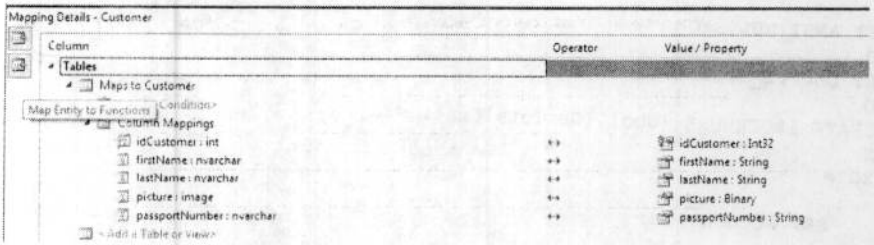


Figura 2.52.- Detalles de mapeo

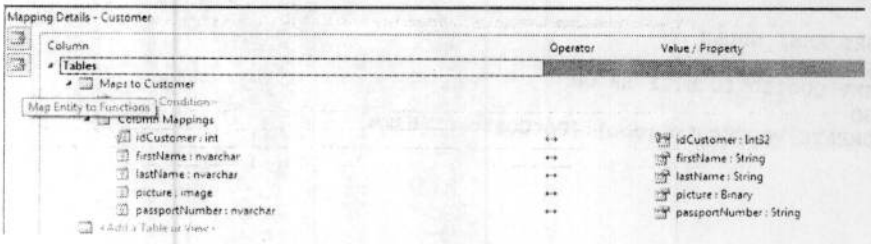


Figura 2.53.- Mapeo de las funciones de una entidad

Una vez asignados los procedimientos almacenados a cada función, tendremos que ir relacionando cada parámetro de los procedimientos con las columnas de nuestras entidades, si los nombres de los parámetros fueran iguales que las propiedades se inferirían automáticamente, pero, en nuestro caso, esto no es así.

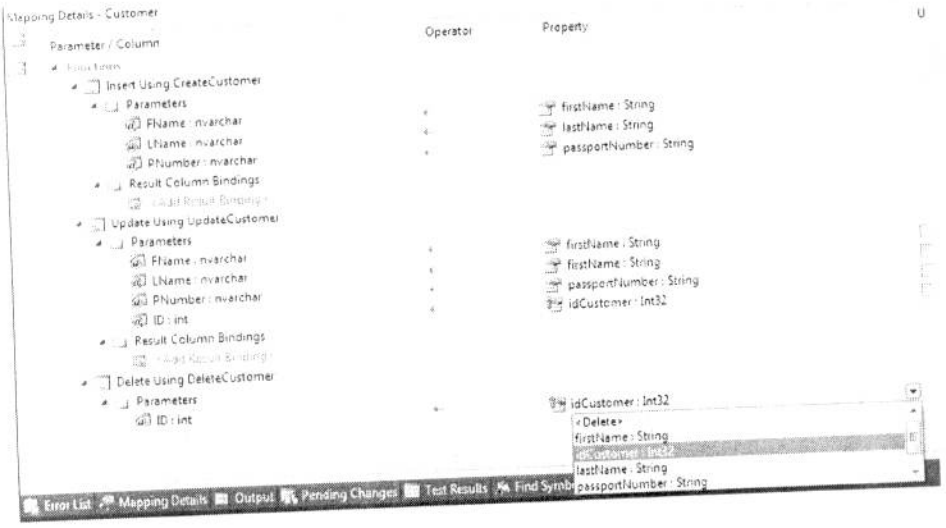


Figura 2.54.- Detalles de mapeo

La otra vía de uso de procedimientos almacenados lógicamente es la que nos permite ejecutar consultas en nuestras bases de datos, cuyos valores de vuelta pueden ser escalares, entidades dentro del modelo, otros tipos o nada. Para ver las distintas opciones crearemos distintos procedimientos almacenados, en concreto los que se pueden ver a continuación.

```
--SQL\EF4_EX_12.sql

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[GetTotalCustomers]
AS
BEGIN

    SET NOCOUNT ON;

  SELECT COUNT(*) from dbo.Customer
END
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[GetCustomersName]
```

```
AS
BEGIN

    SET NOCOUNT ON;

  SELECT idCustomer,firstName from dbo.Customer
END
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[GetCustomersByName]
    @FName nvarchar(200)
AS
BEGIN

    SET NOCOUNT ON;

  SELECT * from dbo.Customer WHERE firstName like @FName
END
GO
```



Figura 2.55.- Creando una función de importación

Ahora que tenemos definidos los procedimientos en nuestra base de datos, actualizaremos el modelo de nuestro último ejemplo e incorporaremos los tres nuevos elementos *GetCustomersByName*, *GetCustomersName* y *GetTotalCustomers*.

Empezaremos a mostrar su uso con el primero. Para ello, nos dirigiremos al *navegador del modelo o model browser* y en la sección de procedimientos almacenados seleccionaremos el que nos ocupa y mostraremos su menú contextual, figura 2.55.

De este menú contextual seleccionaremos la opción *"Crear función de importación"*, acción que nos mostrará un pequeño asistente como el de la figura 2.56.

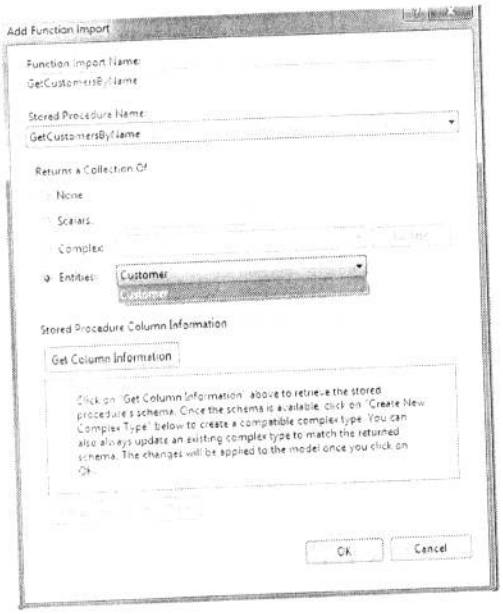


Figura 2.56.- Asistente de creación de función de importación

Por medio de este asistente, como se observa también en la figura anterior, podremos seleccionar el tipo de resultados del procedimiento almacenado, en nuestro caso una entidad *Customer*, y el nombre que queramos que tenga en el modelo. Una vez hecho este sencillo paso, la función ya estaría en disposición de ser consumida, esto lo veremos en los siguientes capítulos.

El segundo de nuestros procedimientos almacenados, *GetCustomesName*, está creado a propósito para devolver un conjunto de columnas que no se corresponden realmente con una entidad completa. Nuestro propósito es ver cómo podemos crear un nuevo tipo, complejo, para dar cabida a estos resultados.

Realizamos los mismos pasos que en el procedimiento anterior hasta mostrar el asistente, en éste, como se ve en la figura 2.57, tenemos la opción de mostrar la información de retorno del procedimiento almacenado, en nuestro caso dos columnas llamadas *idCustomer* y *firstName*.

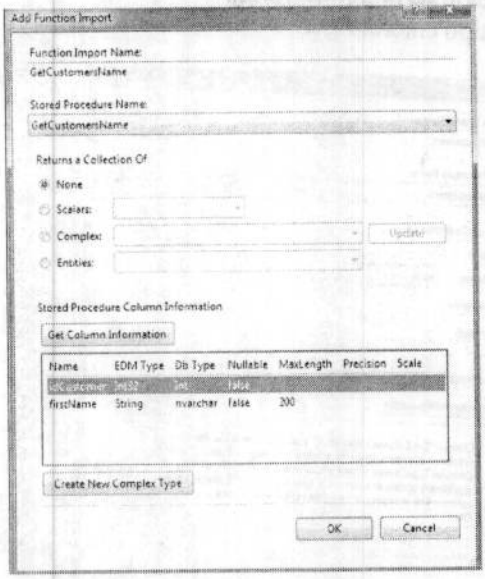


Figura 2.57.- Creando una función de importación con tipos complejos

Puesto que en el modelo no tenemos ninguna entidad capaz de representar este resultado, el mismo asistente nos da la posibilidad de crear un nuevo tipo complejo y hacer que sus resultados se establezcan en él, figura 2.58.

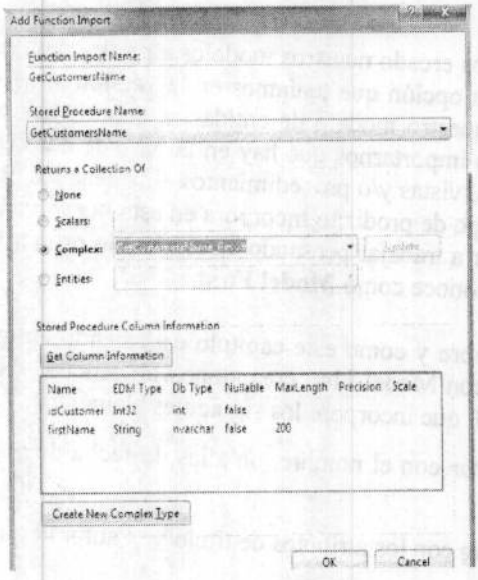


Figura 2.58.- Creando el tipo complejo para una función

El último de los ejemplos es el más sencillo de todos, *GetTotalCustomers* nos devuelve un número entero representando al número de elementos en la tabla *Customers*. Para crear su función de importación, solamente tenemos que asignarle a su tipo de retorno un escalar de tipo entero, figura 2.59.

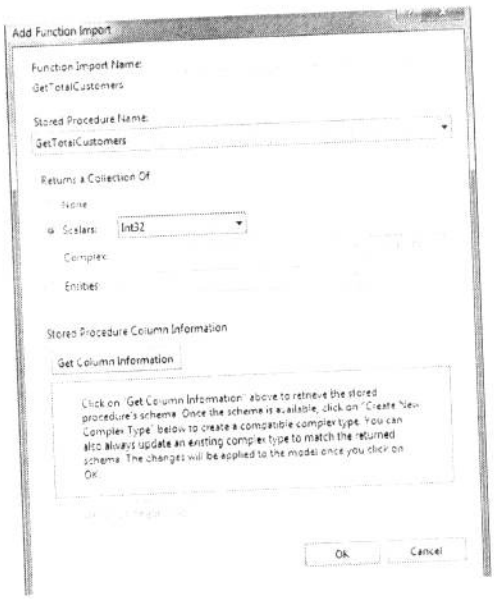


Figura 2.59.- Tipo escalar para una función

1.2.- Primero el modelo por favor.

Hasta ahora siempre hemos creado nuestros modelos a partir de un esquema relacional, de hecho esta era la única opción que teníamos en la versión anterior. No obstante, si de verdad pensamos en nuestro modelo de entidades, deberíamos tener la opción de crear nuestro dominio sin importarnos qué hay en la base de datos, y sin necesidad de tener un soporte de tablas, vistas y/o procedimientos almacenados por detrás, ¿verdad?. Pensando en esto, el equipo de producto incorpora en esta nueva versión de EF, EF 4.1, la posibilidad de empezar a trabajar pensando en el modelo primero, característica que en la literatura de EF se conoce como **Model First**.

Por no perder costumbre y como este capítulo está siendo eminentemente práctico veremos cómo trabajar con Model First creándonos un sencillo modelo de gestión de blogs, **code\EF4_EX_13**, que incorpore los siguientes elementos:

- Una entidad *Autor* con el nombre, un alias, la fecha de alta y un identificador único.
- Una entidad *Blog* con los atributos de título y el autor al que pertenece.

- Una entidad *Post* con los atributos título, contenido y fecha de publicación así como una colección de comentarios asociados.

Una vez definido lo que necesitamos en nuestro dominio empezaremos por crear un proyecto nuevo e incorporar un nuevo modelo de entidades a éste, sin seleccionar la opción de crearlo a partir de una base de datos sino vacío, como en la figura 2.60.

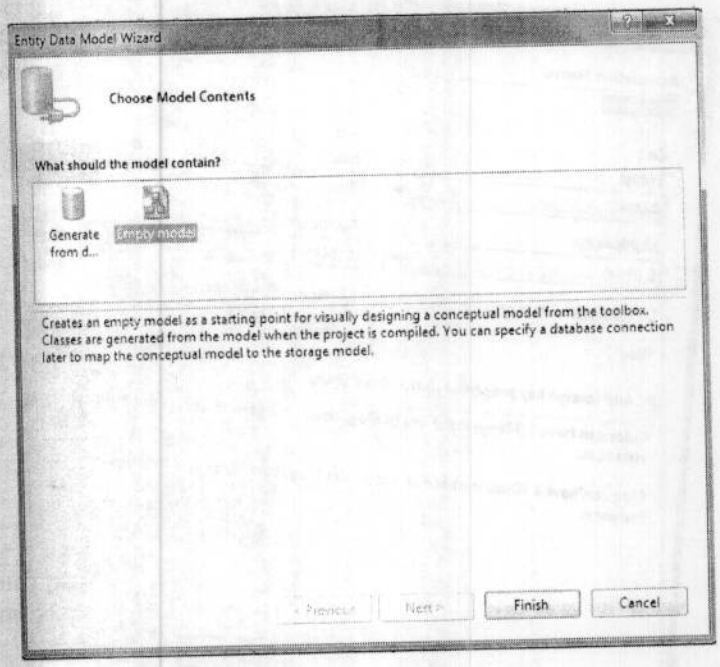


Figura 2.60.- Un modelo con Model First

En esta superficie en blanco empezaremos por agregar una nueva entidad de nuestros requerimientos, *Autor* por ejemplo. Para ello podemos hacer uso de la ventana de herramientas o de la opción de agregar en el menú contextual. A esta nueva entidad le añadiremos las siguientes propiedades:

Tabla 3.- Propiedades de la entidad Autor

Propiedad	Tipo
Name	String
Alias	String
CreationDate	DateTime
IdAutor	Integer

Una vez creada la entidad anterior crearemos la entidad *Blog* con las siguientes propiedades:

Tabla 4- Propiedades de la entidad Blog

Propiedad	Tipo
IdBlog	Integer
Title	String

Puesto que por nuestros requerimientos sabemos que hay una relación entre *Autor* y *Blog*, directamente en nuestro modelo crearemos una nueva asociación, la cual nos permitirá seleccionar los extremos de la misma, la cardinalidad y otros atributos importantes como se puede ver en la figura 2.61.

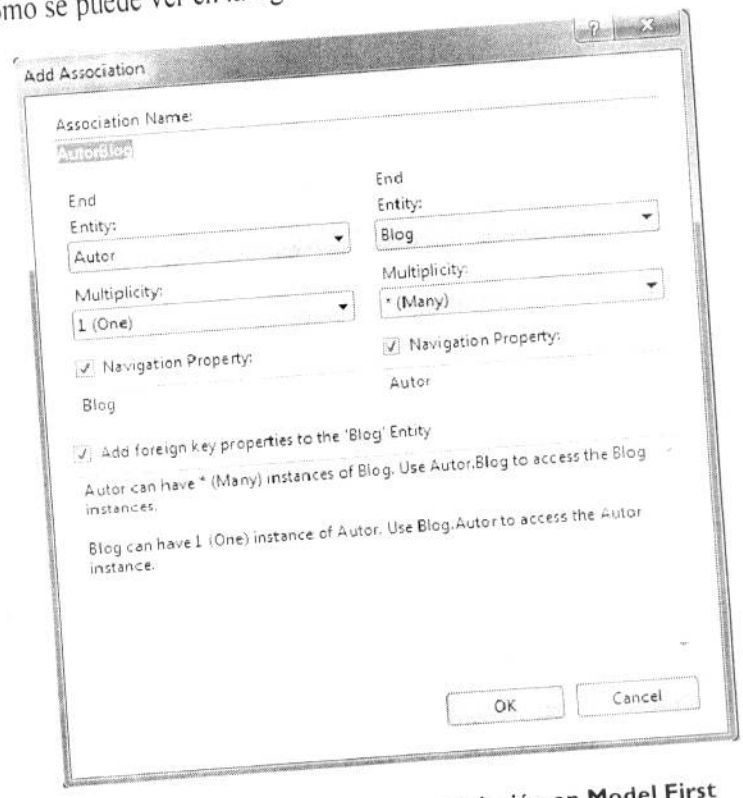


Figura 2.61.- Creación de una asociación en Model First

Si dejamos seleccionada la opción de generar propiedades de clave externa veremos como adicionalmente dentro de nuestra entidad *Blog*, EDM habrá creado una nueva propiedad llamada *AuthorIdAuthor*. Ésta, como se imaginará, representa el valor de la clave del *Autor* asociado. El nombre dado es una convención del nombre de la entidad y el nombre de la clave. Por supuesto usted puede renombrar dicha propiedad.

Pasamos entonces a nuestra entidad *Post*, la cual contendrá las propiedades de la siguiente tabla y una asociación con *Blog* similar a la creada en el paso anterior. El conjunto hasta este momento se muestra en la figura 2.62.

Tabla 5.- Propiedades de la entidad Post

Propiedad	Tipo
IdPost	Integer
BodyContent	String
Title	String
CreationDate	DateTime
IdBlog	Integer

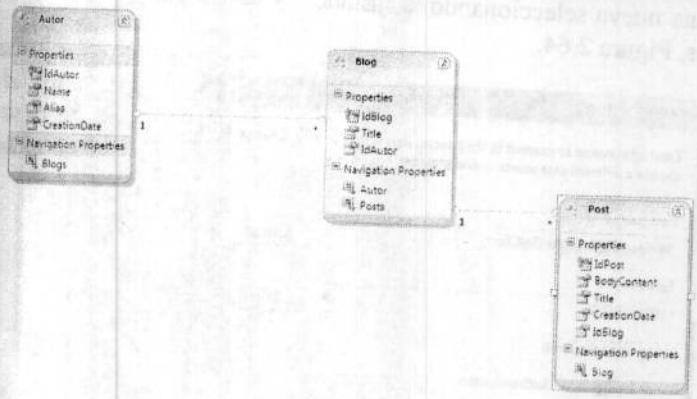


Figura 2.62.- Modelo hasta la creación de Post

El ultimo pasó será incluir la entidad de comentarios, paso que dejamos al lector como ejercicio, puesto que prácticamente es una repetición de lo anterior.

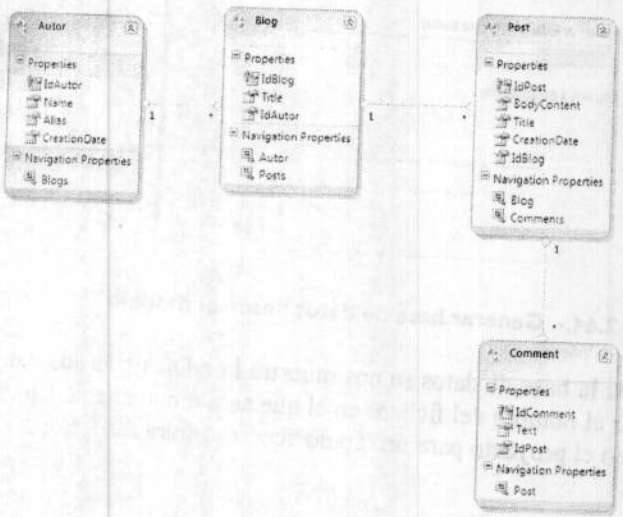


Figura 2.63.- Modelo completado

Fíjese como ahora, al haber seleccionado empezar por un modelo vacío, aunque tengamos las entidades sin mapear, podremos compilar o validar el modelo sin obtener advertencias ni errores por la falta de mapeo.

Lógicamente en algún momento tendremos que especificar contra que tablas, vistas u otros artefactos mapeamos. Para ello Entity Data Model pone a nuestra disposición la posibilidad de generar automáticamente el DDL de una base de datos que se corresponda con el modelo de entidades. Como entenderá esto tiene distintas implicaciones, pero por ahora, simplemente haremos un pequeño ejemplo. Más adelante discutiremos algunos detalles que le vendrán a la cabeza.

Dentro del menú contextual de EDM disponemos de una opción, que ya habrá visto, llamada *"Generate database from model o Generar base de datos desde el modelo"*. Esta opción nos mostrará en primer lugar el asistente de selección de base de datos. Nosotros crearemos una nueva seleccionando la instancia y escribiendo el nombre del catálogo que queramos, Figura 2.64.

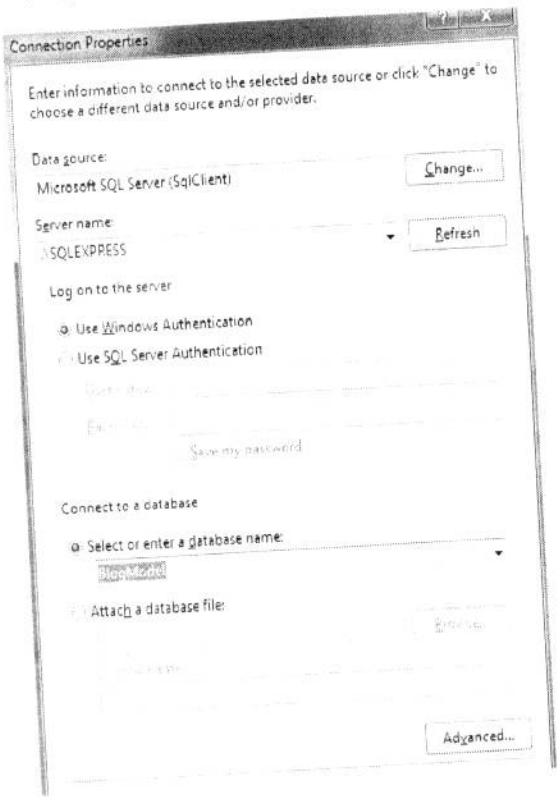


Figura 2.64.- Generar base de datos desde el modelo

Una vez seleccionada la base de datos se nos muestra la DDL generada así como la posibilidad de modificar el nombre del fichero en el que será almacenada. Lógicamente este fichero se incluye en el proyecto para un rápido acceso, figura 2.65.

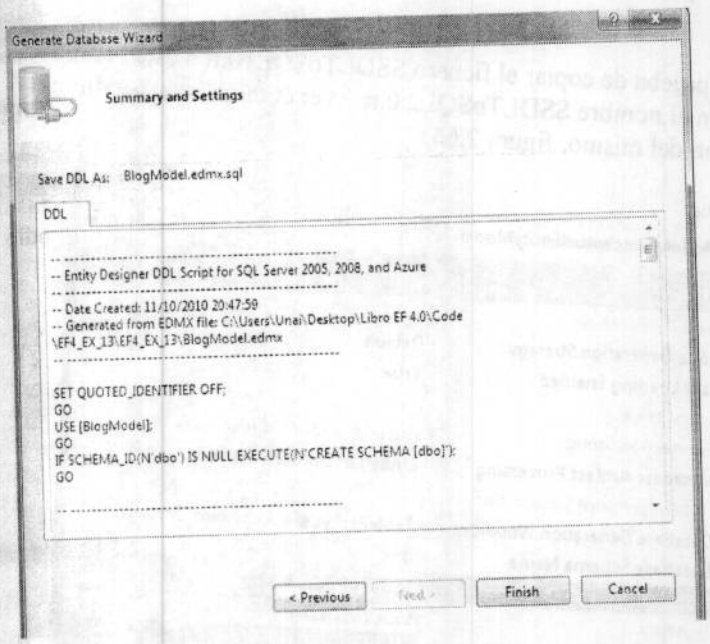


Figura 2.65.- DDL generado a partir del modelo

Completado el asistente, tal y como comentábamos, se incorpora al proyecto un nuevo fichero con la DDL. Además el modelo de **EDM** se modificó para incorporar en los detalles de mapeo de cada tabla y asociación los elementos generados. Es decir, si mostramos la ventana de detalles de mapeo para nuestra entidad *Autor* veremos como ésta ya tiene incluido el *mapping* con una tabla incluida en el fichero **.sql** generado. Es más que probable que se esté preguntando dónde están las distintas convenciones de nombres a utilizar a la hora de generar la DDL, o que estrategias se podrían seleccionar en el caso de tener herencia. Para responder a estas, y también a otras preguntas, podemos ver como en las propiedades de un diseñador de EDM podemos encontrar los siguientes atributos:

- **Database Generation Workflow:** Nos permite seleccionar un flujo de trabajo definido en Windows Workflow Foundation 4.1 con la estrategia a utilizar para generar la base de datos. Por defecto solamente tenemos una denominada *TablePerTypeStrategy.xaml* la cual -como se estará imaginando- define cómo TPT en la base de datos una herencia del modelo de entidades.

- **DDL Generation Template:** Representa la plantilla T4 a utilizar para generar la DDL del fichero .sql de salida. Por defecto solamente tenemos a nuestra disposición *SSDLToSQL10.tt*

Por suerte, tanto el flujo de generación como la plantilla T4 de generación de DDL son adaptables y EDM soporta la selección de otros elementos nuevos. Para ello

solamente tendremos que situar nuevos flujos de generación o plantillas T4 en el siguiente directorio:

[Visual Studio 10.0]\ Common7\IDE\Extensions\Microsoft\Entity Framework Tools\DBGen

Puede hacer la prueba de copiar el fichero SSDLToSQL10.tt a este mismo directorio, pero con el nombre SSDLToSQL2.0.tt y ver como Visual Studio nos permite la selección del mismo, figura 2.66.

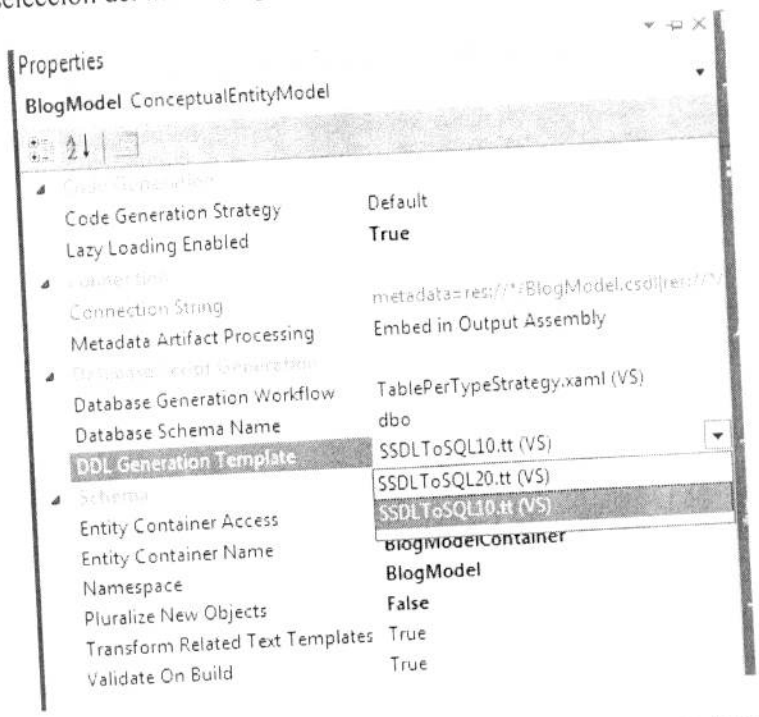


Figura 2.66.- Selección de una plantilla de generación de DDL

Como se imaginará, crear una nueva plantilla de generación de DDL o modificar el workflow de generación es un trabajo que lleva su tiempo, además de implicar conocimientos tanto en T4 como en *Workflow Foundation*. Por suerte, dentro de los complementos que tenemos en el *Extension Manager* de Visual Studio 2010 se encuentra el **"Entity Designer Database Generation Pack"**, figura .2.67, gracias al cual tendremos nuevas opciones de flujos de generación y plantillas de T4, además de otra funcionalidad interesante que describiremos a continuación.

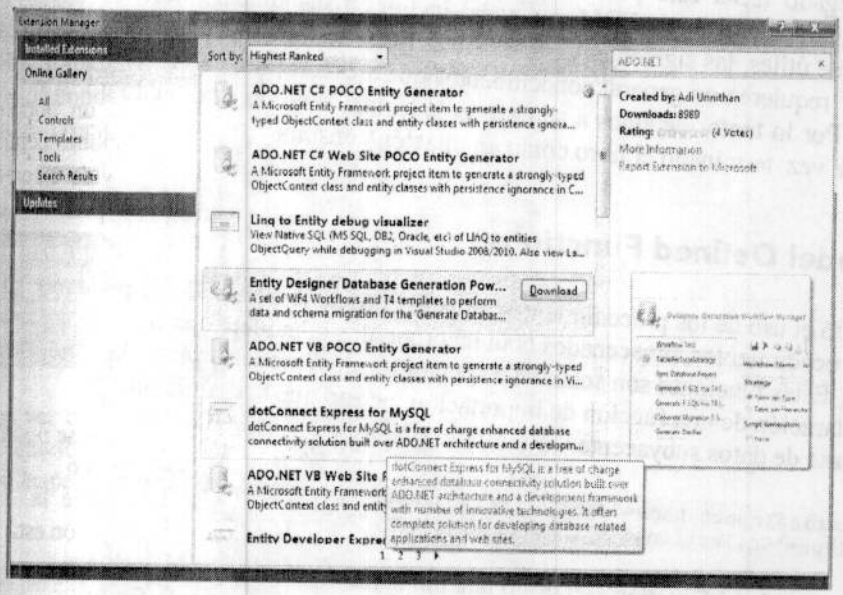


Figura 2.67.- Entity Designer Database Generation Pack

Una vez instalado este Power Pack el asistente de creación de DDL será sustituido por uno nuevo y mucho más potente, gracias al cual podremos realizar las siguientes acciones:

- Seleccionar distintos mecanismos de herencia a la hora de genera la base de datos:
  - Herencia por jerarquía
  - Herencia por tipo
- Generar paquetes de migración
- Generar paquetes de DacPac para Sql Server 2008 R2[5]

Como se habrá dado cuenta, las opciones de este Power Pack son más que interesantes, puesto que más allá de resolver la casuística de usar un u otro workflow de generación, también dispone de opciones para sincronizar una base de datos después de hacer un cambio en un modelo, algo que cuando empezara a trabajar echaría mucho de menos.

[5] Puede obtener más información sobre los paquetes DACPAC en **http://www.microsoft.com/sqlserver/2008/en/us/r2.aspx**

1.3.- Conceptos avanzados de EDM

Si usted ha llegado hasta este punto seguramente habrá descubierto un porcentaje altísimo de las características de EDM que podrá incluir en sus aplicaciones centradas en datos. Aunque útiles, las siguientes características menos habituales y de un carácter más avanzado, requieren en general conocimientos que a estas alturas no tendría usted por qué tener. Por lo tanto dejamos a su criterio la lectura o no de este punto en este momento o una vez terminado el libro como ampliación/consulta.

1.3.1.- Model Defined Functions

Cuando se trató el uso de los procedimientos almacenados dentro de EDM se pudo ver como estos procedimientos almacenados eran incorporados al modelo como *"Funciones"*. Estas funciones son solamente declaradas en el modelo, véase la siguiente declaración de una función de importación, puesto que la definición está hecha en la base de datos subyacente.

```
<FunctionImport Name="GetCustomersByName" EntitySet="Customers"
ReturnType="Collection(EF4_EXModel.Customer)">
```

Una *Model Defined Function (MDF)* o *función definida en el modelo* nos va a permitir saltar la limitación anterior y que podamos tanto declarar como definir una función directamente en Entity Data Model. Lógicamente estas funciones tienen una serie de condicionantes que debemos de conocer antes de mostrar un ejemplo de uso:

- Puesto que se definen en el modelo, el lenguaje a utilizar es Entity SQL.
- Las funciones pueden tomar o no parámetros, pero tienen que tener un tipo de retorno.
- Los parámetros de una función, si los tiene, son solamente de tipo IN. De hecho al contrario que las funciones en SSDL no se permite establecer el modo (*Mode In* o *Mode Out*) de los mismos.
- Tantos los parámetros como el tipo de retorno tiene que ser alguno de los siguientes:
    - Un tipo escalar
    - Un tipo complejo definido en el modelo
    - Una entidad definida en el modelo
    - Una colección de entidades o tipos complejos definidos en el modelo.

Bien, después de esta pequeña introducción pasaremos a realizar algunos ejemplos sencillos. Empezaremos creando un nuevo proyecto, **code\EF4_EX_14**, con un modelo de entidades vacío en la que incluiremos una entidad *Product* con las siguientes propiedades:

Tabla 6.-Propiedades de la entidad Product

Propiedad	Tipo
IdProduct	Integer
Name	String
PubDate	DateTime

Una vez creado el modelo, figura 2.68, procederemos como se ha explicado en el punto anterior a generar la base de datos correspondiente, con lo cual, ya tenemos nuestro mapeo realizado.

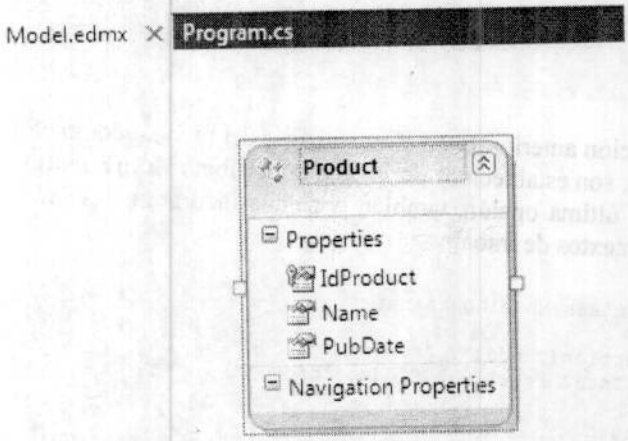


Figura 2.68.- Modelo del ejemplo

El primer paso para crear una función del modelo será el de abrir nuestro fichero *edmx* con un editor *xml*, algo que también hemos hecho por ejemplo cuando se habló de *QueryViews*. Una vez ya tenemos delante nuestro fichero en formato xml, localizaremos el espacio dedicado a nuestro modelo conceptual o *CSDL* e incluiremos dentro del elemento *Schema* lo siguiente:

```
<Function Name="PublishedSince" ReturnType="Edm.Int32">
          <Parameter Name="product" Type="Model.Product"/>
          <DefiningExpression>
            Year(CurrentDateTime()) - Year(product.PubDate)
          </DefiningExpression>
</Function>
```

Como observará, la función introducida llamada *PublishedSince* no es más que una sencilla expresión que toma como parámetro de entrada uno de tipo *Product* y que devuelve un escalar, en concreto el definido en *Edm* como Int32. La definición de la función, tal y como comentábamos antes está expresada en *eSQL*.

Ahora, una vez creada una función del modelo, la pregunta es averiguar cómo se usa. Para ello tenemos distintas vías que iremos viendo. El primer ejemplo de uso de una función del modelo será directamente en *eSQL*. Así, podríamos escribir una consulta tal y como se ve en el siguiente ejemplo:

```
"SELECT VALUE p FROM ModelContainer.Products as p WHERE
Model.PublishedSince(p) > 0 ";
```

Por suerte, el equipo de trabajo de ADO.NET Entity Framework también nos permite el uso de estas *MDF* de forma tipada en nuestras consultas *de Linq To Entities*.

Para ello solamente tendremos que definir una clase como sigue:

```
public static class CustomFunctions
{
    [EdmFunction("Model", "PublishedSince")]
    public static Int32 YearSince(Product producto)
    {
        throw new NotSupportedException("La has cagado");
    }

}
```

Fíjese que en la definición anterior, tanto el *namespace* del modelo como el nombre de la función en el mismo, son establecidos por medio del atributo **EdmFunction**.

Para terminar y como última opción, también podemos incorporar las *MDF* como miembros de nuestros contextos de trabajo:

```
public partial class ModelContainer
{
        [EdmFunction("Model", "PublishedSince")]
        public Int32 YearSince(Product product)
        {
            return
this.QueryProvider.Execute<Int32>(Expression.Call(Expression.Constant
(this),

(MethodInfo)MethodInfo.GetCurrentMethod(),

Expression.Constant(product, typeof(Product))));
        }

}
```

Como era de esperar, además de permitirnos la creación de MDF personalizadas, ADO.NET Entity Framework también dispone de un conjunto de funciones definidas a distintos niveles, como las encontradas en las clases ***EntityFunctions*** y ***SqlFunctions***.

La primera, *EntityFunctions*, nos aporta un buen número de funciones para usar en nuestras consultas, algunas muy interesantes como por ejemplo *AsNonUnicode* la cual nos permite enviar a la base de datos una cadena de caracteres como *No Unicode* . En *SqlFunctions* encontraremos la definición de nuestras funciones favoritas de *Sql Server* listas para usar en nuestras consultas de *L2E* por ejemplo.

1.3.2.- Tabla por tipo concreto y herencia

En puntos anteriores vimos cómo trabajar con herencia dentro de nuestros modelos de entidades. En concreto vimos como mapear y trabajar las jerarquías en el caso en el que nuestros modelos relacionales estuvieran como *TPH* o *TPT*. Repase los puntos anteriores si lo considera necesario antes de continuar.

En los modelos relacionales, además de tener una tabla con un discriminador, *TPH*, o una tabla con datos comunes y una tabla para cada tipo particular relacionadas 1..1 con la común, *TPT*, también existe otra posibilidad de *'simular'* la herencia. Esta tercera vía, conocida como *TPC*, que consiste en disponer de una tabla para cada tipo incluyendo los datos comunes a los demás, no está soportada directamente por *EDM* y por lo tanto requiere de bastante trabajo en el editor *xml*, algo que por otra parte es bastante tedioso.

A lo largo de este punto, mostraremos un ejemplo, **code\EF4_EX_15**, de mapeo *TPC* partiendo del siguiente modelo relacional:

```
--SQL\EF4_EX_13.sql

GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Software](
    [idSoftware] [int] NOT NULL,
    [Title] [nvarchar](200) NOT NULL,
    [Description] [nvarchar](200) NOT NULL,
    [LicenseCode] [nvarchar](20) NULL,
 CONSTRAINT [PK_Software] PRIMARY KEY CLUSTERED
(
    [idSoftware] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Book](
    [IdBook] [int] NOT NULL,
    [Title] [nvarchar](200) NULL,
    [Description] [nvarchar](200) NOT NULL,
```

```
    [ISBN] [nvarchar](15) NULL,
CONSTRAINT [PK_Book] PRIMARY KEY CLUSTERED
(
    [IdBook] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO
```

Creamos un nuevo proyecto de *Visual Studio 2010* que incluya un modelo de entidades partiendo de las tablas anteriores, figura 2.69. El objetivo, dadas las dos entidades presentes será el de tratar de mover los elementos comunes de ellas a una nueva entidad, y después hacer que tanto *Software* como *Book* sean subclases de ella.

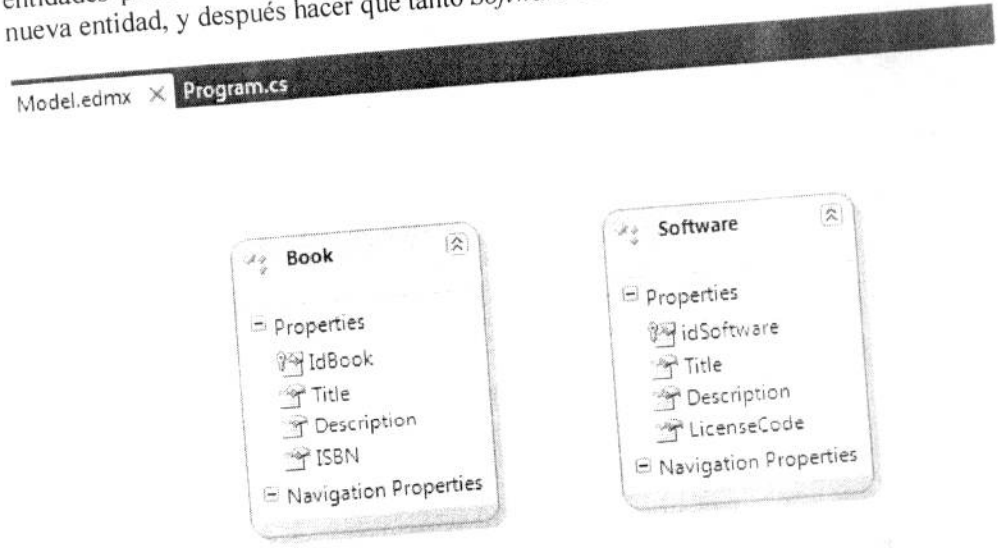


Figura 2.69.- Modelo antes de TPC

La creación de una nueva entidad dentro de un modelo es algo que ya hemos visto, basta con utilizar la ventana de herramientas. Una vez creada la nueva entidad, que llamaremos *Product* podemos cortar las propiedades comunes de una entidad y moverlas hasta ella, de tal forma que el resultado sea como el siguiente, figura 2.70.

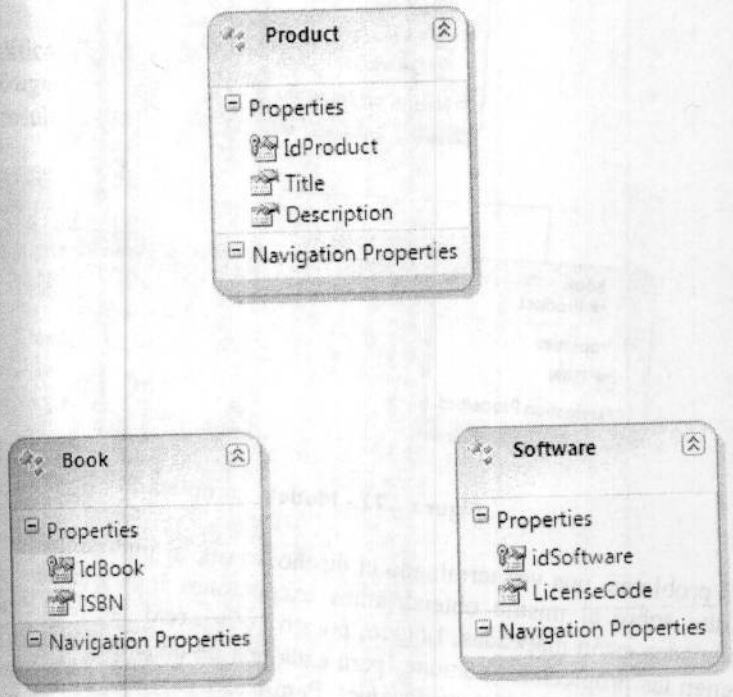


Figura 2.70.- Creando la entidad base

Puesto que nuestra nueva entidad, *Product*, no tendrá una correspondencia real en el modelo relacional ni puede existir como instancia en si misma, tendremos que marcar su propiedad **Abstract** a *true*, figura 2.71.

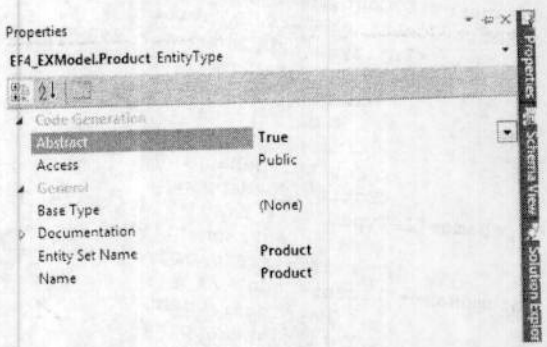


Figura 2.71.- Estableciendo la entidad como abstracta

Una vez hecho este trabajo, ya podemos hacer nuestra relación de herencia, tal y como se ha hecho en más ocasiones a lo largo de este capítulo, al final, tendremos un modelo como el mostrado a continuación, figura 2.72.

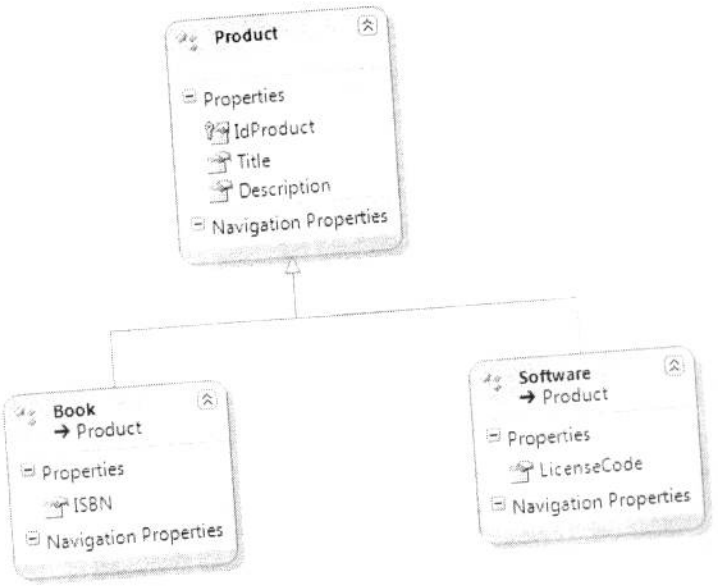


Figura 2.72.- Modelo completado

El problema, una vez terminado el diseño, es que si intentáramos realizar cualquier consulta sobre el mismo obtendríamos excepciones indicándonos que no todas las propiedades están mapeadas. Lógico, puesto que en realidad no hemos dicho cómo se obtienen las propiedades comunes para cada una de las entidades *Software* y *Book*, las cuales ahora están situadas en *Product*. Para hacer este trabajo, como ya comentamos, tendremos que recurrir a editar el fichero *edmx* en formato *xml* y modificar la sección de mapeo, que por defecto contiene lo siguiente:

```
      <Mapping Space="C-S"
xmlns="http://schemas.microsoft.com/ado/2008/09/mapping/cs">
        <EntityContainerMapping
StorageEntityContainer="EF4_EXModelStoreContainer"
CdmEntityContainer="EF4_EXEntities">
          <EntitySetMapping Name="Product">
            <EntityTypeMapping TypeName="IsTypeOf(EF4_EXModel.Book)">
              <MappingFragment StoreEntitySet="Book">
                <ScalarProperty Name="ISBN" ColumnName="ISBN" />
              </MappingFragment>
            </EntityTypeMapping>
            <EntityTypeMapping
TypeName="IsTypeOf(EF4_EXModel.Software)">
              <MappingFragment StoreEntitySet="Software">
                <ScalarProperty Name="LicenseCode"
ColumnName="LicenseCode" />
              </MappingFragment>
            </EntityTypeMapping>
          </EntitySetMapping>
        </EntityContainerMapping>
      </Mapping>
```

La modificación de esta sección consistirá en agregar a cada uno de los elementos *MappingFragment* las propiedades que faltan a cada entidad. Una vez terminado el trabajo el resultado deberá ser igual al mostrado a continuación:

```
<Mapping Space="C-S"
xmlns="http://schemas.microsoft.com/ado/2008/09/mapping/cs">
        <EntityContainerMapping
StorageEntityContainer="EF4_EXModelStoreContainer"
CdmEntityContainer="EF4_EXEntities">
          <EntitySetMapping Name="Product">
            <EntityTypeMapping TypeName="IsTypeOf(EF4_EXModel.Book)">
              <MappingFragment StoreEntitySet="Book">
                <ScalarProperty Name="ISBN" ColumnName="ISBN" />
                <ScalarProperty Name="Title" ColumnName="Title" />
                <ScalarProperty Name="Description"
ColumnName="Description" />
                <ScalarProperty Name="IdProduct" ColumnName="IdBook"
/>
              </MappingFragment>
            </EntityTypeMapping>
            <EntityTypeMapping
TypeName="IsTypeOf(EF4_EXModel.Software)">
              <MappingFragment StoreEntitySet="Software">
                <ScalarProperty Name="LicenseCode"
ColumnName="LicenseCode" />
                <ScalarProperty Name="Title" ColumnName="Title" />
                <ScalarProperty Name="Description"
ColumnName="Description" />
                <ScalarProperty Name="IdProduct"
ColumnName="idSoftware" />
              </MappingFragment>
            </EntityTypeMapping>
          </EntitySetMapping>
        </EntityContainerMapping>
    </Mapping>
```

1.4.- Dividir lo grande

Este es uno de los puntos que más dudas generó con respecto a su inclusión en el libro. En primer lugar porque es de un gran carácter teórico, algo que no se corresponde con el libro ya que trata de ser siempre lo más práctico posible con ejemplos varios. Y en segundo lugar porque tratarlo en profundidad requeriría casi un libro entero. De hecho, ya existe una magnífica obra de referencia que puede consultar, **"Domain Driven Design" de Eric Evans.**

A pesar de todo, la experiencia nos dice que siempre que se habla de ADO.NET Entity Framework, sale a la luz el trabajo con modelos de grandes dimensiones o cómo dividir en modelos cuando detrás tenemos una base de datos con una ingente cantidad de tablas. Al final tanto para modelos EDM como para otras casuísticas el problema suele ser el mismo: cómo trabajar con lo grande o *como dividir lo grande*, si esto tiene sentido.

En aplicaciones de medio-gran tamaño el número de elementos presente en un modelo de EDM crece a una gran velocidad en términos de elementos y relaciones entre los mismos. Mantener la coherencia en modelos grandes es muy complicado debido no solo al tamaño, sino a que sobre los mismos suele estar trabajando al mismo tiempo un gran número de personas. La verdad es que aunque la idea de mantener un modelo único podría llegar a resultarle tentadora, realmente no resulta muy viable y a su vez es demasiado costosa en términos de productividad.

Una pregunta que nos podríamos hacer es si de verdad necesitamos integración entre todas y cada una de las funcionalidades de nuestro sistema. En otras palabras: ¿podemos construir verticales de nuestra solución?.

La respuesta a esta pregunta suele ser que **si** en la mayoría de las veces, por lo tanto podríamos pensar en dividir un **modelo grande** en **submodelos** más pequeños que contengan subconjuntos del principal. Por supuesto, un objetivo fundamental es que mantengamos la coherencia, dentro de un dominio, de cada uno de los *submodelos*. No debemos pensar solamente en una partición fundamental.

Ahora, una vez que hemos decidido que podemos y debemos dividir lo grande, la pregunta es qué estrategia o estrategias podemos utilizar para hacerlo. Podríamos pensar en dividir modelos en función de los equipos de trabajo que tengamos, por funcionalidades del sistema etc.

1.4.1.- Relaciones entre contextos

Las distintas relaciones que se pueden dar entre distintos contextos dependen fundamentalmente del grado de comunicación que exista entre los distintos equipos de cada contexto y del grado de control que se tenga de los mismos. Por ejemplo, podría ocurrir que no podamos realizar modificaciones en un contexto, como puede ser el casi de un sistema *legacy*. O podría ocurrir también que nuestro sistema necesitase de otros para funcionar. A continuación veremos algunas relaciones que se dan entre los distintos contextos, submodelos.

Shared Kernel

Cuando tenemos dos o más equipos que pueden comunicarse de forma fluida, es interesante que estos puedan tener responsabilidades compartidas sobre sus modelos de trabajo, es decir, que los equipos involucrados puedan tener entidades comunes definidas en los distintos modelos. Estos objetos pasan a formar lo que se denomina **Shared Kernel** o núcleo compartido de ambos contextos, y se suele dejar establecido que para realizar cualquier modificación de objetos en el *Shared Kernel* es necesaria la aprobación de todos los equipos que lo comparten. En este escenario cualquier estrategia que favorezca la comunicación entre equipos, como por ejemplo pudiera ser la rotación de los miembros de los mismos, es por supuesto bien recibida.

Customer/ Supplier

Es bastante frecuente encontrar un sistema que depende de otros sistemas para funcionar, como por ejemplo podrían ser sistemas de análisis o de toma de decisiones. Estas dependencias entre los contextos, submodelos, son en una sola dirección, desde el sistema dependiente hacia el sistema del que se depende. Con este tipo de relaciones entre los contextos, al igual que anteriormente, la comunicación entre los miembros de los diferentes contextos es necesaria puesto que el contexto proveedor suele necesitar funcionalidad del contexto cliente y habitualmente dentro del equipo de trabajo del contexto que ofrece una funcionalidad la frecuencia de cambio es pequeña por miedo a afectar a quienes lo consumen.

Conformista

La relación cliente/proveedor requiere de la colaboración entre los equipos de los distintos contextos. Esta situación suele ser bastante ideal, y en la mayoría de los casos el contexto proveedor tiene sus propias prioridades y no está dispuesto a atender a las necesidades del contexto cliente. En este tipo de situaciones donde nuestro contexto depende de otro contexto sobre el cual no tenemos control alguno (no podemos realizar modificaciones ni pedir funcionalidades), y con el que tenemos una estrecha relación (el coste de la traducción de las comunicaciones de un contexto a otro es elevado), podemos emplear un acercamiento conformista. Éste consiste en acomodar nuestro modelo al expuesto por el otro contexto. Esto limita nuestro modelo a hacer simples adiciones al modelo del otro contexto, y limita la forma que puede tomar nuestro modelo. No obstante no es una idea descabellada, ya que posiblemente el otro modelo incorpore el conocimiento acumulado en el desarrollo del otro contexto. La decisión de seguir una relación de conformismo depende en gran medida de la calidad del modelo del otro contexto.

Anti-corruption layer

La mayoría de las relaciones entre *submodelos* que hemos visto hasta ahora presuponen una buena comunicación entre los equipos de trabajo que los gestionan. Lógicamente también se presupone la bondad de los contextos con los que nos relacionamos. En alguna ocasión puede que nos tengamos que enfrentar al hecho de que nos debemos relacionar con un contexto realizado por un equipo con el que no tenemos comunicación (puede que ni los conozcamos), y que además ese modelo no esté "muy bien realizado". Para este tipo de soluciones podemos implementar un **anti-corruption layer**, que consiste en crear una capa intermedia entre contextos que se encargue de realizar traducciones entre nuestro contexto y los contextos con los que nos

relacionamos. De forma general un *anti-corruption layer* se compone de tres componentes fundamentales, los *adaptadores*, los *traductores* y las *fachadas*. Primero se diseña la fachada que simplifique la comunicación, frente a esta fachada se colocan los adaptadores que permiten adaptar la interfaz de un contexto al otro. Por último también tenemos los traductores capaces de traducir elementos de un *submodelo* a elementos de otro *submodelo*.

Separate-ways

Separate-ways en realidad es una manera de no relacionar. Es decir, las funcionalidades se hacen en *submodelos* distintos por equipos distintos sin necesidad de comunicación entre ellos. Si tuviéramos funcionalidades que necesitaran hacer uso de otros contextos se delegarían en orquestaciones de más alto nivel.

CAPÍTULO 3

Entity Client

1.- INTRODUCCIÓN

En el capítulo anterior hemos visto cómo crear nuestros modelos conceptuales a partir de distintos modelos físicos de bases de datos, o bien partiendo de cero gracias a la nueva característica de **Model First** en *ADO.NET EF 4.1*. Los modelos conceptuales conforman el segundo de los tres espacios que se definen dentro de *EF*. El primero lo determina el modelo físico subyacente, es decir, el conjunto de tablas, vistas, procedimientos almacenados, etc., consultable por la vía tradicional que nos propone ADO.NET desde sus primeras versiones. El segundo espacio, el espacio conceptual, hemos visto cómo construirlo a partir del modelador de entidades, pero hasta ahora no sabemos nada sobre cómo consultarlo. A lo largo de este capítulo nos centraremos en *Entity Client*, la primera de las posibilidades disponibles a la hora de consultar los modelos de entidades.

Nota: Este capítulo no es esencial para el lector. Realmente usted puede empezar a trabajar con ADO.NET EF 4.1 sin necesidad de conocer Entity Client. Se presenta en el libro para aquellas personas que quieran conocer más acerca de las interioridades de Entity Framework.

Los **ejemplos** de este capítulo tienen un fin eminentemente demostrativo, no pretenden ser una guía y por lo tanto su calidad no está tan cuidada como los que encontraremos en el siguiente capítulo. Todos ellos se basan en el modelo relacional y los datos que puede encontrar con los materiales adjuntos a este libro, en concreto los ficheros sql:

a) mamazon_0_2_model.sql

b) mamazon_0_2_data.sql

1.1.- Entity Client como proveedor de ADO.NET

Desde la versión 2.0 de .NET Framework, ADO.NET pone a nuestra disposición una arquitectura de acceso a datos basada en la herencia que hace posible que podamos practicar una programación uniforme independientemente del proveedor específico de bases de datos que vayamos a utilizar. El espacio de nombres `System.Data.Common` contiene la definición de las clases base fundamentales de las que se hereda luego en los distintos proveedores específicos, como `SqlClient` u `OracleClient` (proveedor éste obsoleto en la última versión). En la siguiente imagen podemos ver algunas de las clases base de esta jerarquía.

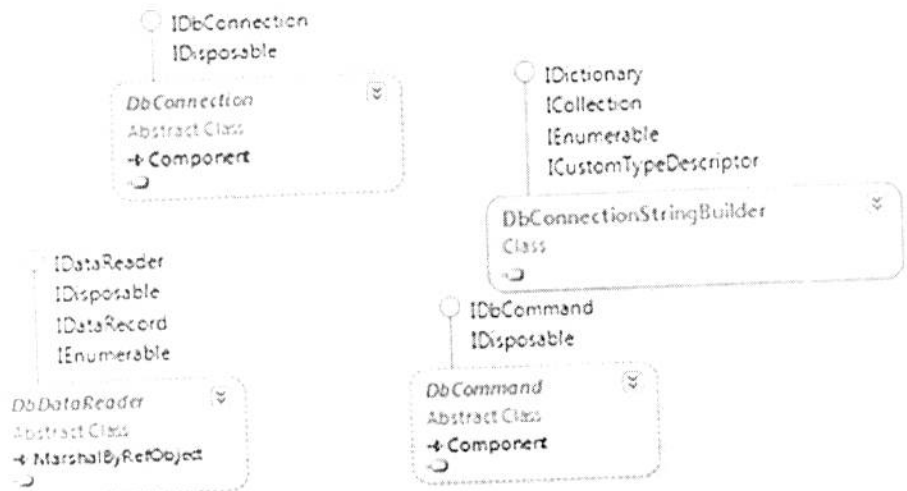


Figura 3.1.- Jerarquía de ADO.NET

La llegada del Marco de entidades trae aparejada consigo la aparición de un nuevo proveedor de ADO.NET, **Entity Client,** que nos permitirá trabajar sobre modelos EDM de una manera muy similar a como lo hacíamos anteriormente con otros proveedores como `SqlClient`. La principal diferencia entre este nuevo proveedor y el resto de los disponibles radica en qué se consulta y en el lenguaje utilizado para consultar. Entity Client tiene como único objetivo la consulta de nuestros modelos conceptuales, los modelos EDM que hemos aprendido a construir en el capítulo anterior.

Y puesto que uno de los objetivos fundamentales de la tecnología es ser 'ignorante' del sistema de bases de datos físico que estemos usando, no tendría sentido utilizar alguno de los dialectos específicos existentes en la actualidad, como Transact SQL, PL-SQL, etc. Por ello, además de las clases que conforman este nuevo proveedor, se ha desarrollado un nuevo dialecto de consultas "neutral" denominado **Entity SQL** (abreviadamente, **eSQL**), dialecto sobre el que se ofrece una amplia referencia en uno de los apéndices de este libro. *Entity Client* se encarga de traducir las sentencias eSQL a sentencias en el dialecto específico de la base de datos subyacente.

Nota: En realidad la sentencias eSQL se traducen inicialmente a una representación intermedia conocida como Canonical Query Trees (CQT), y luego a partir de esta se genera el SQL específico del almacén.

En la siguiente tabla podemos ver las clases implementadas en este nuevo proveedor y sus correspondientes ancestros dentro de la jerarquía de `System.Data.Common`.

Tabla 1.- Relación de Entity Client con la jerarquía de proveedores

Clase de Entity Client	Base en la jerarquía de proveedores
EntityConnectionStringBuilder	DbConnectionStringBuilder
EntityConnection	DbConnection
EntityCommand	DbCommand
EntityDataReader	DbDataReader
EntityParameter	DbParameter
EntityParameterCollection	DbParameterCollection
EntityTransaction	DbTransaction
EntityProviderFactory	DbProviderFactory

Tal y como hemos comentado, existe una perfecta relación entre las clases de este nuevo proveedor y las clase base de la tabla anterior; lo cual es totalmente lógico, puesto que el objetivo fundamental de esta jerarquía de proveedores es precisamente ofrecer un patrón Factoría con el que los programadores puedan trabajar. En las siguientes secciones iremos desglosando cada una de estas clases, mostrando el objetivo y la funcionalidad de cada una de ellas.

1.1.1.- EntityConnectionStringBuilder

Al igual que su clase base, `DbConnectionStringBuilder`, esta clase nos brinda la posibilidad de crear cadenas de conexión de una forma fuertemente tipada. A pesar de ser una clase a menudo olvidada, `DbConnectionStringBuilder` y todas las clases que heredan de ella nos permiten de una forma muy sencilla generar una cadena de conexión correcta para el proveedor específico con el que estemos trabajando, sin necesidad de acordarnos explícitamente de cómo se forma la cadena, y por supuesto con la ventaja de no cometer errores al entrecomillarla, algo bastante usual por otra parte.

Las cadenas de conexión en *Entity Client* se componen de tres partes fundamentales:

- Los **metadatos**, que nos permiten especificar la localización del modelo conceptual con el que vamos a trabajar; es decir, las secciones SSDL, CSDL y MSL.
- El **proveedor de ADO.NET** específico a utilizar para acceder a la base de datos subyacente al modelo.
- La cadena de conexión para dicho proveedor subyacente.

`EntityConnectionStringBuilder` nos permite representar estos tres elementos por medio de distintas propiedades, tal y como se puede ver en el siguiente listado, que contiene parte de la definición de esta clase.

```
public sealed class EntityConnectionStringBuilder :
    DbConnectionStringBuilder
{
   public EntityConnectionStringBuilder();
   public EntityConnectionStringBuilder(
      string connectionString);

   public string Metadata { get; set; }
   public string Provider { get; set; }
   public string ProviderConnectionString { get; set; }
}
```

La sección de metadatos es la única realmente nueva para nosotros, puesto que todo lo que tiene que ver con la conexión a la base de datos subyacente es algo que damos por conocido de anteriores versiones de ADO.NET. Como ya hemos mencionado, los metadatos nos permiten especificar la ubicación física de los archivos SSDL, CSDL y MSL de definición del modelo. El diseñador de entidades integrado en Visual Studio 2010 nos permite de una manera muy sencilla indicar la situación de estos elementos mediante la ventana de propiedades del modelo conceptual (figura 3.2). Esta situación puede ser un determinado directorio físico o, de manera alternativa, estos ficheros pueden ser embebidos como recursos dentro del ensamblado en el que se aloja el modelo. Ésta última es la opción por defecto.

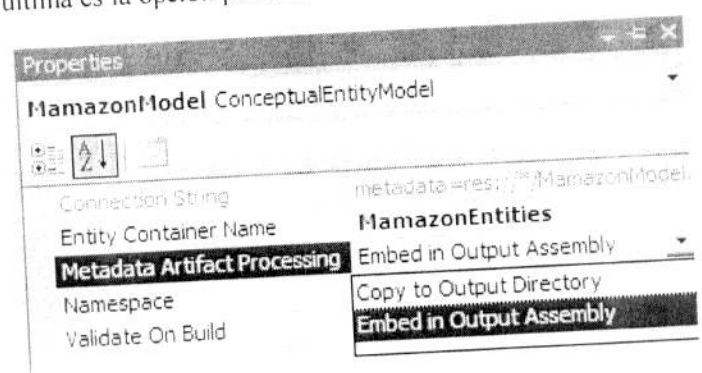


Figura 3.2.- Selección de la localización del modelo

En caso de mantener seleccionada la opción por defecto, si revisamos el ensamblado con alguna herramienta como ILDASM o Reflector podremos ver las tres secciones del modelo como recursos incrustados (figura 3.3).

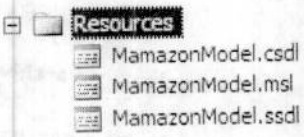


Figura 3.3.- Las secciones del modelo como recursos incrustados

En este caso, en la cadena de conexión correspondiente los metadatos se reflejan mediante una expresión similar a la siguiente:

```
metadata=res://*/MamazonModel.csdl|res://*/MamazonModel.ssdl|
res://*/MamazonModel.msl
```

Aquí el prefijo **res** indica que esos elementos se encuentran incluidos como recursos. A continuación se especifican las rutas de cada una de las secciones, separadas por barras verticales.

Nota: Recordemos que si define el modelo dentro de una librería de clases, llamémosle A, y utiliza ese modelo desde cualquier otra librería o ejecutable, la ruta de los recursos incrustados debe incluir el camino hacia el ensamblado A. Para nuestro modelo de ejemplo la especificación correcta sería la siguiente.

metadata=res://A/MamazonModel.csdl|
res://A/MamazonModel.msl|res://A/MamazonModel.ssdl

Si por el contrario, tomamos la decisión de desplegar las secciones del modelo como archivos en disco, al lado del ensamblado de definición del modelo encontraremos tres archivos XML, uno por cada sección del modelo (figura 3.4).

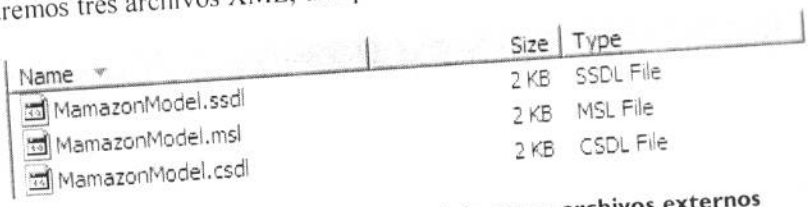


Figura 3.4.- Secciones del modelo como archivos externos

Ahora los metadatos constan únicamente de las rutas hacia los tres archivos de secciones del modelo:

```
metadata=.\MamazonModel.csdl|.\MamazonModel.ssdl|.\MamazonModel.msl
```

Nota:
Recuerde que la ruta .\ siempre hace referencia **al punto de entrada del ejecutable** y no a la ruta del ensamblado que aloja el modelo

Puesto que ya sabemos cómo especificar la localización del modelo, ya podremos utilizar `EntityConnectionStringBuilder`. Las siguientes líneas de código hacen uso de esta clase para especificar la cadena de conexión de *Entity Client* a un modelo, denominado `MamazonModel`, que trabaja con una base de datos de *SQL Server*.

```
static void Main(string[] args)
{
   EntityConnectionStringBuilder connectionStringBuilder =
     new EntityConnectionStringBuilder();

   // localización de las secciones del modelo
   connectionStringBuilder.Metadata =
     @"res://*/MamazonModel.csdl|
     res://*/MamazonModel.ssdl|res://*/MamazonModel.msl";

   // el proveedor de la base de datos subyacente
   connectionStringBuilder.Provider =
     "System.Data.SqlClient";

   // la cadena de conexión a la base de datos
   connectionStringBuilder.ProviderConnectionString =
     @"Server=.\SQLEXPRESS;Initial Catalog=Mamazon;
Integrated Security=true;MultipleActiveResultSets=true";
   // ...
}
```

Nota: Debe de incluir una referencia al ensamblado System.Data.Entity para poder utilizar las clases de EntityClient. Una vez agregada la referencia importe el espacio de nombres System.Data.EntityClient.

1.1.2.- Entity Connection

Como su propio nombre indica, esta clase al igual que su homóloga en la jerarquía, representa una conexión; en este caso, al modelo conceptual. Para realizar cualquier trabajo sobre una fuente de datos con *ADO.NET*, lo primero que necesitamos es una conexión a la que pedirle que ejecute las distintas acciones que queramos ejecutar sobre la fuente de datos. Para poder construir y abrir una conexión contra un modelo de *EDM* necesitamos una cadena de conexión, algo que ya hemos aprendido a hacer. Una vez que la conexión conoce la información del modelo y del proveedor con el que va a trabajar, podemos abrirla y llamar a los distintos métodos de trabajo que la misma nos ofrece.

Las conexiones dentro de *Entity Client* se representan mediante instancias de la clase `EntityConnection`, cuya estructura básica puede verse a continuación.

```
public sealed class EntityConnection : DbConnection
{
   public EntityConnection();
   public EntityConnection(string connectionString);
   public EntityConnection(MetadataWorkspace workspace,
     DbConnection connection);

   public override string ConnectionString { get; set; }
   public override int ConnectionTimeout { get; }
   public override string Database { get; }
   public override string DataSource { get; }

   protected override DbProviderFactory
      DbProviderFactory { get; }
   public override string ServerVersion { get; }
   public override ConnectionState State { get; }
   public DbConnection StoreConnection { get; }

   protected override DbTransaction
      BeginDbTransaction(IsolationLevel isolationLevel);
   public EntityTransaction BeginTransaction();
   public EntityTransaction BeginTransaction(
      IsolationLevel isolationLevel);

   public override void ChangeDatabase(string databaseName);

   public EntityCommand CreateCommand();
   protected override DbCommand CreateDbCommand();

   public override void Close();
   protected override void Dispose(bool disposing);

   public MetadataWorkspace GetMetadataWorkspace();
```

```
    public override void Open();
}
```

Además de los distintos métodos y propiedades de su clase base, `DbConnection`, `EntityConnection` define algunas propiedades adicionales, como `StoreConnection` (la conexión a la base de datos subyacente), o `MetadataWorkspace`, clase que encapsula la información de las distintas secciones del modelo y metadatos, y algunas redefiniciones de métodos virtuales provenientes de su clase base.

La preparación de una nueva instancia de `EntityConnection` es muy sencilla: basta con inicializar, mediante la propiedad `ConnectionString` o mediante el propio constructor de la clase, la cadena de conexión a utilizar. El listado siguiente nos muestra cómo crear y abrir una nueva instancia de `EntityConnection` a partir de la cadena de conexión definida anteriormente.

```
static void Main(string[] args)
{
   // Creación de instancia de EntityConnectionStringBuilder
   EntityConnectionStringBuilder connectionStringBuilder =
      new EntityConnectionStringBuilder();

   // Localización de las secciones del modelo
   connectionStringBuilder.Metadata =
     @"res://*/MamazonModel.csdl|
     res://*/MamazonModel.ssdl|res://*/MamazonModel.msl";

   // Proveedor de la base de datos subyacente
   connectionStringBuilder.Provider = "System.Data.SqlClient";

   // Cadena de conexión a la base de datos
   connectionStringBuilder.ProviderConnectionString =
      @"Server=.\SQLEXPRES;Initial Catalog=Mamazon;
Integrated Security=true;MultipleActiveResultSets=true";

   EntityConnection connection = new EntityConnection(
      connectionStringBuilder.ConnectionString);

   try
   {
      connection.Open();
   }
   catch (FileNotFoundException)
   {
      // La información de metadatos no está expresada
      // correctamente, no se pueden localizar las secciones
      // SSDL, CSDL o MSL
      // TODO: hacer trabajo aquí
   }
   catch (ArgumentException)
   {
      // La propiedad Provider no está establecida
      // correctamente
      // TODO: hacer trabajo aquí
   }
```

```
        catch (EntityException)
        {
            // Se ha producido un error y no se puede
            // conectar con la base de datos subyacente
            // TODO: hacer trabajo aquí
        }
    }
```

1.1.3.- EntityConnection y EntityDataReader

Una vez que ya conocemos cómo establecer una conexión a un modelo de entidades mediante las clases del proveedor *Entity Client*, solamente nos queda ver cómo realizar consultas a las distintas entidades y funciones que podremos tener dentro del modelo.

Como podrá imaginar el lector si ha trabajado alguna vez con *ADO.NET*, la clase `EntityCommand` nos permite especificar el comando que deseamos ejecutar sobre la conexión contra la que el comando trabaje. Para ello, la clase dispone de las propiedades `CommandText`, mediante la cual estableceremos nuestra consulta en *eSQL*, y `CommandType`, con la que podremos especificar el tipo de comando a ejecutar. Una vez construida una instancia de `EntityCommand`, ya sea por la vía tradicional o mediante una llamada al método `CreateCommand` de la clase `EntityConnection`, podremos obtener un lector del conjunto de datos resultante de la consulta con una simple llamada a `ExecuteDataReader`.

En las siguientes líneas de código se muestra un ejemplo de uso de la clase `EntityCommand` y de cómo obtener nuestro objeto de lectura. En este caso, la consulta a ejecutar es una sentencia simple en eSQL que recupera la lista de entidades `Audio` de nuestra base de datos `Mamazon`. Por ahora no nos centraremos en el proceso de lectura; en este caso el proceso no es tan simple, ni tan potente, como al que estamos acostumbrados de versiones anteriores de *ADO.NET*.

Nota: Le recordamos que tanto los esquemas como los datos de la base de datos Mamazon pueden ser obtenidos en la carpeta sql de los materiales que se adjuntan con este libro.

```
static void Main(string[] args)
{
    // Creación de instancia de EntityConnectionStringBuilder
    EntityConnectionStringBuilder connectionStringBuilder =
        new EntityConnectionStringBuilder();

    // Localización de las secciones del modelo
    connectionStringBuilder.Metadata =
        @"res://*/MamazonModel.csdl|
        res://*/MamazonModel.ssdl|res://*/MamazonModel.msl";

    // Proveedor de la base de datos subyacente
    connectionStringBuilder.Provider = "System.Data.SqlClient";

    // Cadena de conexión a la base de datos
```

```
    connectionStringBuilder.ProviderConnectionString =
       @"Server=.\SQLEXPRESS;Initial Catalog=Mamazon;
Integrated Security=true;MultipleActiveResultSets=true";

    EntityConnection connection = new EntityConnection(
       connectionStringBuilder.ConnectionString);

    try
    {
       connection.Open();

       // Creamos un objeto de conexión
       EntityCommand command = connection.CreateCommand();
       command.CommandText =
          "SELECT VALUE t FROM MamazonEntities.Audio AS t";
       command.CommandType = CommandType.Text;

       // Obtenemos el  lector de la consulta
       EntityDataReader reader =
         command.ExecuteReader(CommandBehavior.SequentialAccess);

       while (reader.Read())
       {
          // Hacer el trabajo de lectura
       }

       reader.Close();
    }
    catch (FileNotFoundException)
    {
       // La información de metadatos no está expresada
       // correctamente, no se pueden localizar las secciones
       // SSDL, CSDL o MSL
       // TODO: hacer trabajo aquí
    }
    catch (ArgumentException)
    {
       // La propiedad Provider no está establecida
       // correctamente
       // TODO: hacer trabajo aquí
    }
    catch (EntityException)
    {
       // Se ha producido un error y no se puede
       // conectar con la base de datos subyacente
       // TODO: hacer trabajo aquí
    }
    finally
    {
       connection.Close();
    }
}
```

Si revisa el código anterior, podrá comprobar cómo efectivamente este proceso es prácticamente idéntico al modelo de programación que teníamos anteriormente. La principal diferencia la veremos en la siguiente sección: realizar la lectura de los datos obtenidos por medio de la consulta realizada. Hasta ahora, nuestra visión de los datos obtenidos por medio de una consulta constaba únicamente de filas que contenían un

número determinado de columnas que procedíamos a leer por medio de los distintos métodos `Get` que el objeto `DbDataReader` nos proporcionaba. En el caso de *Entity Client*, el conjunto de resultados de una determinada consulta no tiene por qué ser tan simple como un conjunto de filas con un conjunto de columnas por cada una.

Suponga que disponemos de un modelo de *EDM* con dos entidades como `Cliente` y `Pedido`, con una relación uno a muchos entre ellas. Hasta ahora, si queríamos obtener los datos de clientes y sus pedidos con una consulta tradicional, el conjunto de resultados adolecía de la repetición de los datos en muchas de las columnas obtenidas para cada una de las filas, como se muestra en la siguiente tabla de ejemplo.

IDCLIENTE	NOMBRE	IMPORTE	FECHA
1	UNAI	120	01-01-2008
1	UNAI	200	01-03-2006
2	OCTAVIO	220	02-04-2007
2	OCTAVIO	150	02-03-2004

Dentro de los modelos de *EDM* disponemos, como hemos podido ver en el capítulo anterior, de las **propiedades de navegación**, gracias a las cuales este tipo de consultas se facilitan en gran medida. En realidad, solicitar estos datos será tan simple como realizar una consulta semejante a:

```
SELECT c, c.Pedido FROM MamazonEntities.Cliente AS c
```

Pero el conjunto de resultados que se obtiene como resultado de ejecutar una consulta como la anterior dista mucho del conjunto de resultados de una consulta "tradicional"; en este caso, *Entity Client* conoce la estructura del conjunto de resultados a obtener, y es capaz de envolver los resultados en sus tipos correspondientes, sin necesidad de darnos valores repetidos que tengamos que tener en cuenta para una lectura correcta de los mismos. La siguiente tabla podría representar el resultado de la ejecución de nuestra consulta en *eSQL*.

C		C.PEDIDO	
IDCLIENTE		IMPORTE	FECHA
1		120	01-01-2008
		200	01-03-2006
IDCLIENTE	NOMBRE	IMPORTE	FECHA
2	OCTAVIO	220	02-04-2007
		150	02-03-2007

Como puede observar, cada uno de los registros de salida representa una estructura totalmente diferente a la de una 'fila' convencional. En nuestro caso, la salida de la consulta anterior consta de una entidad `Cliente` que contiene un conjunto de entidades

`Pedido`, los pedidos de ese cliente. Seguramente ya se habrá dado cuenta de que la complejidad de la lectura de estas estructuras de datos aumenta considerablemente, pero no se asuste; en éste y posteriores capítulos veremos cómo este proceso queda reducido a la más absoluta sencillez gracias al uso de *Object Services* y *LINQ to Entities*.

Por supuesto, existen otros muchos ejemplos que nos demuestran las diferencias entre este tipo de proceso de lectura y el de aquellas a las que estábamos acostumbrados. En la siguiente la tabla se presenta la estructura de los resultados que produce la consulta:

```
SELECT REF(c),c.Pedido FROM MamazonEntities.Cliente AS c
```

C		C.PEDIDO	
ENTITYSET	IDCLIENTE	IMPORTE	FECHA
CLIENTE	1	120	1.
		200	01-03-2006
ENTITYSET	IDCLIENTE	IMPORTE	FECHA
CLIENTE	2	220	02-04-2007
		150	02-03-2007

Una vez que ya conocemos que el proceso de lectura es diferente al proceso tradicional al que estábamos acostumbrados, y que *Entity Client* es capaz de envolver estos resultados en objetos, nuestro principal propósito debe concentrarse en entender qué tipos de datos nos podemos encontrar en las lecturas y cómo tratarlos. Esto es precisamente lo que veremos en la siguiente sección de este capítulo.

1.2.- Trabajando con Entity Client

1.2.1.- Jerarquía de tipos

Cualquiera de los elementos que podemos obtener como resultado de consultar nuestro modelo pertenece a uno de los tipos de la jerarquía impuesta por `MetadataItem`, del espacio de nombres `System.Data.Metadata.Edm`, cuya estructura básica se presenta a continuación.

```
public abstract class MetadataItem
{
   public abstract BuiltInTypeKind BuiltInTypeKind { get; }

   public Documentation Documentation { get; set; }

   public ReadOnlyMetadataCollection<MetadataProperty>
```

```
        MetadataProperties { get; }

    public static EdmType GetBuiltInType(
        BuiltInTypeKind builtInTypeKind);

    public static ReadOnlyCollection<FacetDescription>
        GetGeneralFacetDescriptions();
}
```

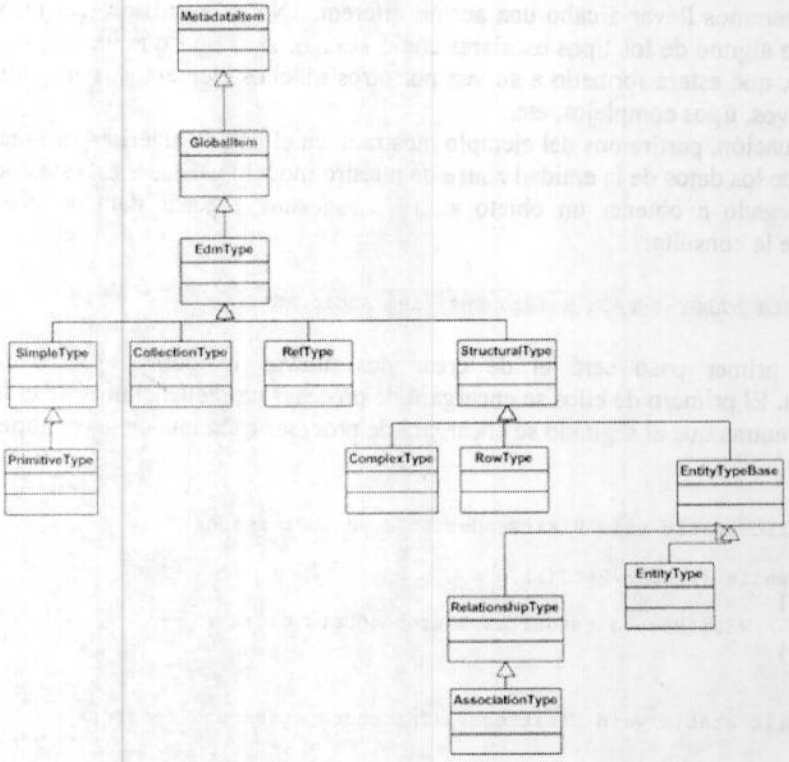


Figura 3.5.- Jerarquía de metadata Item

De entre los principales miembros de esta clase base abstracta, podemos destacar el enumerado `BuiltInTypeKind`, mediante el cual podremos acceder a un valor que nos indica a cuál de los tipos pertenece en concreto el elemento, de entre los distintos tipos posibles con los que podemos trabajar, algunos de los cuales se muestran en el listado siguiente.

```
public enum BuiltInTypeKind
{
    AssociationType = 3,
    // ...
    CollectionType = 6,
// ...
```

```
    ComplexType = 8,
    // ...
    EntityType = 14,
    // ...
    PrimitiveType = 26,
    // ...
    RefType = 31,
    // ...
    RelationshipType = 35,
    RowType = 36,
    SimpleType = 37,
    StructuralType = 38,
    // ...
}
```

En función del tipo concreto que estemos leyendo, indicado por el enumerado anterior, deberemos llevar a cabo una acción diferente. No es lo mismo leer un valor primitivo -de alguno de los tipos escalares como **string**, **int**, etc.- que un tipo entidad determinado, que estará formado a su vez por otros muchos elementos, como nuevos tipos primitivos, tipos complejos, etc.

A continuación, partiremos del ejemplo mostrado en el listado anterior para realizar una lectura de los datos de la entidad `Audio` de nuestro modelo `Mamazon`. En este código, habíamos llegado a obtener un objeto `EntityDataReader` a partir del cual leer los resultados de la consulta:

```
SELECT VALUE t FROM MamazonEntities.Audio AS t
```

Nuestro primer paso será el de crear dos nuevos métodos, `VisitReader` y `VisitRecord`. El primero de ellos se encargará de procesar un secuencialmente el lector de datos, mientras que el segundo se encargará de procesar cada una de las lecturas que se realicen.

```
public static void VisitReader(DbDataReader reader)
{
    while (reader.Read())
    {
        VisitRecord(reader as IExtendedDataRecord);
    }
}

public static void VisitRecord(IExtendedDataRecord record)
{
    // ...
}
```

Como puede apreciarse en el código anterior, cada una de las lecturas obtenidas por el método `VisitReader` se pasarán al método `VisitRecord`, interpretando el objeto leído como uno que implementa la interfaz `IExtendedDataRecord` de `System.Data`. Cada uno de estos objetos `IExtendedDataRecord` puede además (tal y como vemos en la

definición de esta interfaz, que se muestra en el listado siguiente) darnos acceso a la información del tipo concreto y la obtención de nuevos lectores o registros internos, como ocurre en el caso de la consulta que devuelve, para cada cliente, los pedidos de dicho cliente.

```
public interface IExtendedDataRecord : IDataRecord
{
   DataRecordInfo DataRecordInfo { get; }

   DbDataReader GetDataReader(int i);

   DbDataRecord GetDataRecord(int i);
}
```

Definida ya la organización de nuestros métodos de trabajo, solamente nos queda proporcionar la funcionalidad del método `VisitRecord`. El primer paso a dar será el de examinar cuántos elementos debemos procesar en nuestra lectura, independientemente de que estos elementos sean a su vez tipos simples, estructurados o nuevos lectores. Para esta tarea, simplemente necesitamos obtener el valor de la propiedad `FieldCount` que la interfaz base de `IExtendedDataRecord`, `IDataRecord`, nos provee.

```
int fieldCount = record.FieldCount;
```

Una vez conocido el número de elementos dentro del registro que estamos leyendo, iremos examinando el tipo al que pertenece cada uno de ellos dentro de la jerarquía de `MetadataItem` (figura 3.5); para ello nos serviremos del elemento `FieldMetadata` de cada uno de los elementos del registro actual.

```
BuiltInTypeKind fieldTypeKind =
   record.DataRecordInfo.FieldMetadata[i].FieldType.
      TypeUsage.EdmType.BuiltInTypeKind;
```

Como hemos comentado anteriormente, en dependencia del tipo de cada uno de los elementos del registro la lectura será diferente. No es lo mismo leer un `RefType` que un `ComplexType`, por lo que podemos tomar la decisión de utilizar una sentencia condicional con el fin de invocar al proceso más adecuado.

```
switch (fieldTypeKind)
{
   case BuiltInTypeKind.EntityType:
      break;
   case BuiltInTypeKind.RefType:
      break;
   case BuiltInTypeKind.CollectionType:
      break;
   case BuiltInTypeKind.PrimitiveType:
      break;
   case BuiltInTypeKind.RowType:
      break;
   case BuiltInTypeKind.ComplexType:
      break;
```

```
        default:
            break;
    }
```

El código del lector hasta este momento puede verse en el siguiente listado.

```
public static void VisitRecord(IExtendedDataRecord record)
{
    // Obtenemos el número de elementos a procesar
    int fieldCount = record.FieldCount;
    // Recorremos y procesamos los elementos
    for (int i = 0; i < fieldCount; i++)
    {
        BuiltInTypeKind fieldTypeKind =
            record.DataRecordInfo.FieldMetadata[i].FieldType.
                TypeUsage.EdmType.BuiltInTypeKind;

        switch (fieldTypeKind)
        {
            case BuiltInTypeKind.EntityType:
                break;
            case BuiltInTypeKind.RefType:
                break;
            case BuiltInTypeKind.CollectionType:
                break;
            case BuiltInTypeKind.PrimitiveType:
                break;
            case BuiltInTypeKind.RowType:
                break;
            case BuiltInTypeKind.ComplexType:
                break;
            default:
                break;
        }
    }
}
```

Casi hemos terminado nuestro trabajo; solamente nos queda saber qué hacer cuando nos encontremos con los distintos elementos de la jerarquía en nuestro bloque `switch`. En el caso de los elementos estructurales, `RowType`, `ComplexType` y `EntityType`, estos elementos son a su vez contenedores de otros elementos, simples o estructurales, por lo que el contenido del registro en esa posición es a su vez un nuevo `IExtendedDataRecord` que debemos procesar; o sea, que para los tipos estructurales debemos hacer una llamada recursiva al método `VisitRecord`. En el caso de que en cualquier posición del registro actual encontremos un objeto del tipo `CollectionType`, entonces internamente lo que tendremos que procesar es un lector, un objeto de tipo `DbDataReader`, con lo que este caso también nos lleva a una llamada recursiva, en este caso a `VisitReader`. Los casos `PrimitiveType` y `RefType` son los más sencillos de evaluar; el primero de ellos corresponde a un tipo escalar, por lo que podremos llamar al método `GetValue` del registro para obtener ese valor; en cuanto a `RefType`, el valor que se nos proporcionará será un objeto de tipo `EntityKey`, que hemos visto en el capítulo 2 y que contiene los pares nombre/valor que especifican la(s) clave(s) de la entidad.

El siguiente listado presenta el código completo de nuestro lector, y la figura 3.6 un posible resultado de la consulta.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data.EntityClient;
using System.IO;
using System.Data;
using System.Data.Common;
using System.Data.Metadata.Edm;

namespace PlainConcepts.Book.ParseEntityCommandSample
{
  class Program
  {
    static void Main(string[] args)
    {
      EntityConnectionStringBuilder connectionStringBuilder =
          new EntityConnectionStringBuilder();
      connectionStringBuilder.Metadata =
@"res://*/MamazonModel.csdl|
res://*/MamazonModel.ssdl|res://*/MamazonModel.msl";

      connectionStringBuilder.Provider =
        "System.Data.SqlClient";
      connectionStringBuilder.ProviderConnectionString =
@"Server=.\SQLEXPRESS;Initial Catalog=Mamazon;
Integrated Security=true;MultipleActiveResultSets=true";

      // Creamos un objeto conexión
      EntityConnection connection =
        new EntityConnection(
          connectionStringBuilder.ConnectionString);

      try
      {
        connection.Open();
        // Creamos un objeto comando
        EntityCommand command = connection.CreateCommand();
        command.CommandText =
          "SELECT  t FROM MamazonEntities.Audio AS t";
        command.CommandType = CommandType.Text;
        // Obtenemos el  lector de la consulta
        EntityDataReader reader =
          command.ExecuteReader(
            CommandBehavior.SequentialAccess);
        VisitReader(reader as DbDataReader);
        reader.Close();
      }
      catch (FileNotFoundException)
      {
        // La información de metadatos no está expresada
        // correctamente, no se pueden localizar las secciones
        // SSDL, CSDL o MSL
        // TODO: hacer trabajo aquí
      }
      catch (ArgumentException)
      {
        // La propiedad Provider no está establecida
```

```
        // correctamente
        // TODO: hacer trabajo aquí
      }
      catch (EntityException ex)
      {
        // Se ha producido un error y no se puede
        // conectar con la base de datos subyacente
        // TODO: hacer trabajo aquí
      }
      finally
      {
        connection.Close();
        Console.ReadLine();
      }
    }

    public static void VisitReader(DbDataReader reader)
    {
      while (reader.Read())
      {
        VisitRecord(reader as IExtendedDataRecord);
      }
    }

    public static void VisitRecord(IExtendedDataRecord record)
    {
      // Obtenemos el número de elementos a procesar
      int fieldCount = record.FieldCount;

      for (int i = 0; i < fieldCount; i++)
      {
        BuiltInTypeKind fieldTypeKind =
          record.DataRecordInfo.FieldMetadata[i].FieldType.
            TypeUsage.EdmType.BuiltInTypeKind;
        switch (fieldTypeKind)
        {
          case BuiltInTypeKind.EntityType:
            VisitRecord(
              record.GetDataRecord(i) as IExtendedDataRecord);
            break;
          case BuiltInTypeKind.RefType:
            WriteEntityKey(record.GetValue(i) as EntityKey);
            break;
          case BuiltInTypeKind.CollectionType:
            VisitReader(
              record.GetDataReader(i) as DbDataReader);
            break;
          case BuiltInTypeKind.PrimitiveType:
            WritePrimitiveType(
              record.GetValue(i),
              record.DataRecordInfo.FieldMetadata[i].
                FieldType.Name);
            break;
          case BuiltInTypeKind.RowType:
          case BuiltInTypeKind.ComplexType:
            VisitRecord(
              record.GetDataRecord(i) as IExtendedDataRecord);
            break;
          default:
            Console.WriteLine("Elemento no reconocido");
            break;
```

```
            }
        }
    }

    static void WritePrimitiveType(object value,string name)
    {
        Console.WriteLine("{0}:{1}",name,value);
    }

    static void WriteEntityKey(EntityKey key)
    {
        Console.WriteLine("Tipo EntityKey");
        foreach (EntityKeyMember member in key.EntityKeyValues)
        {
            Console.WriteLine("\t Clave:{0} Valor:{1}",
                member.Key,member.Value);
        }
    }
}
}
```

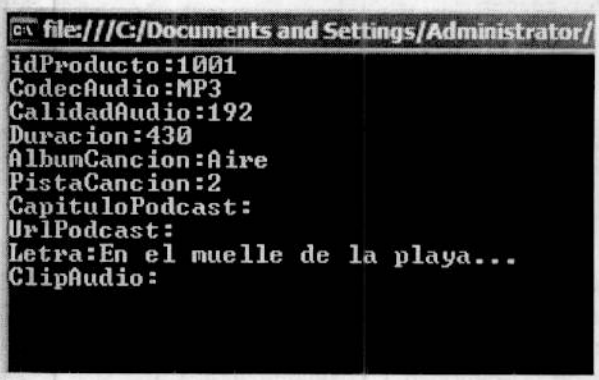


Figura 3.6.- Resultados de la ejecución

1.2.2.- Consultas parametrizadas

Como cabría esperar, al igual que el resto de proveedores de ADO.NET, Entity Client soporta el uso de consultas parametrizadas. Su uso es idéntico al que estamos acostumbrados, aunque con alguna diferencia sin mayor importancia. La clase `EntityParameter` es la que nos suministra la posibilidad de establecer parámetros dentro de nuestras consultas. Para ello únicamente necesitamos crear instancias de esta clase e indicarles el nombre del parámetro, el tipo y el valor que queremos otorgarle.

Con el fin de poner un ejemplo útil, veamos a continuación cómo paginar los resultados de una consulta. Para ello, eSQL pone a nuestra disposición las cláusulas **SKIP** y **LIMIT** (consultar el apéndice de referencia de eSQL), con lo que paginar una consulta resulta especialmente sencillo.

La consulta con paginación en cuestión es la siguiente:

```
    SELECT t FROM MamazonEntities.Audio AS t ORDER BY t.idProducto SKIP
@skip LIMIT @limit
```

Verá que en esta consulta usamos dos parámetros, denominados `@skip` y `@limit`, cuyos valores procederemos a establecer en las siguientes líneas de código.

```
// Creamos un objeto comando
EntityCommand command = connection.CreateCommand();
command.CommandText =
@"SELECT t FROM MamazonEntities.Audio AS t
  ORDER BY t.idProducto SKIP @skip LIMIT @limit";
command.CommandType = CommandType.Text;

EntityParameter skipParameter =
   new EntityParameter("skip", DbType.Int32);
skipParameter.Value = 0;
command.Parameters.Add(skipParameter);

EntityParameter limitParameter =
   new EntityParameter("limit", DbType.Int32);
limitParameter.Value = 10;
command.Parameters.Add(limitParameter);

// Obtenemos el  lector de la consulta
EntityDataReader reader =
   command.ExecuteReader(CommandBehavior.SequentialAccess);
```

Nota: Fíjese como a diferencia de SqlClient el nombre de los parámetros no incluye el símbolo @.

Por supuesto, no vamos hablar aquí de las bondades que ofrece la ejecución de consultas parametrizadas en los proveedores tradicionales de *ADO.NET*. Seguramente el lector ya conozca de sobra todas las ventajas que este tipo de consulta nos aporta desde el punto de vista de la seguridad y el rendimiento. En el caso de *eSQL*, usar consultas parametrizadas implica que la consulta "traducida" para el proveedor de base de datos específico también usará consultas parametrizadas.

1.2.3.- Consultas polimórficas

En el capítulo 2 de este libro, hicimos mucho hincapié en las distintas posibilidades que EDM nos proporciona para el modelado de jerarquías de entidades, presentando las tres posibilidades más comunes dentro de los modelos relacionales: herencia por tipo, herencia por subtipo y herencia por jerarquía. Llegados a este punto, en el que ya sabemos cómo utilizar *Entity Client* para la consulta de nuestros modelos conceptuales, veamos cómo consultar de manera polimórfica estas jerarquías de objetos.

Para este ejemplo, nos serviremos del caso práctico del capítulo 2 denominado 'Herencia por subtipo', en el que construimos un modelo como el que vemos en la figura 3.7.

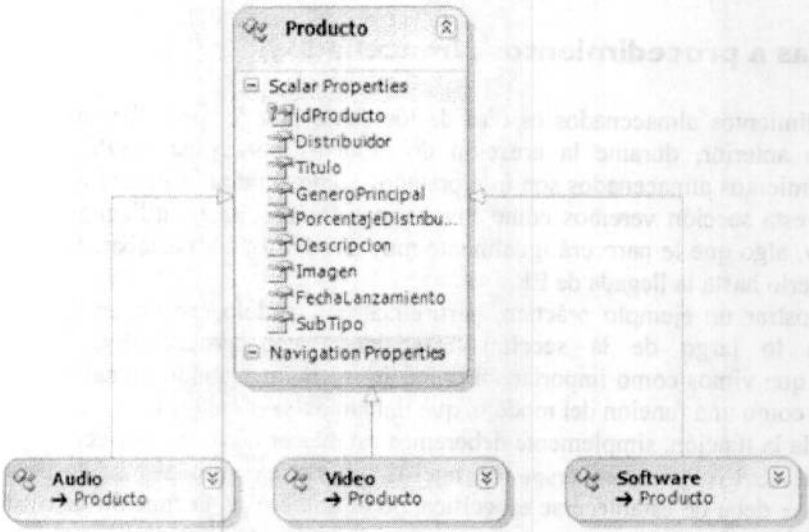


Figura 3.7.- Modelo EDM de nuestro ejemplo

El proceso de lectura a utilizar es idéntico al mostrado en el listado anterior; de hecho, éste es un bloque genérico que siempre podrá reutilizar. Lo que queremos destacar aquí es cómo expresar en eSQL consultas que permitan obtener un conjunto de resultados que contenga únicamente los objetos de alguno de los subtipos de la jerarquía. Para ello, eSQL prevé el uso de la expresión `OFTYPE`, cuya firma es la siguiente:

```
OFTYPE(Expresión, Tipo)
  Expresión: Expresión de consulta que devuelve una colección de
objetos
  Tipo: El tipo contra el que comprobar cada uno de los objetos de
esa colección
```

Para nuestro ejemplo de modelo, si quisiéramos obtener todos los elementos de tipo `Audio` de nuestro modelo `Mamazon`, la consulta que tendríamos que ejecutar sería la siguiente:

```
SELECT audio.Titulo, audio.Duracion
FROM OFTYPE(MamazonEntities.Producto, MamazonModel.Audio) AS audio
```

Otra alternativa a la expresión `OFTYPE` en eSQL es el uso del elemento `TREAT`:

```
TREAT(Expresión AS Tipo)
  Expresión: Expresión de consulta que devuelve una entidad
  Tipo: El tipo (derivado) como el que se desea tratar la expresión
```

Una consulta similar a la anterior, pero basada en el uso de **TREAT** sería la siguiente:

```
SELECT VALUE TREAT(p AS MamazonModel.Audio)
FROM MamazonEntities.Producto AS p WHERE p IS OF(MamazonModel.Audio)
```

1.2.4.- Llamadas a procedimientos almacenados

El uso de procedimientos almacenados es otro de los casos prácticos que hemos visto en el capítulo anterior, durante la creación de modelos conceptuales. Para nosotros, los procedimientos almacenados son interpretados como funciones dentro de nuestro modelo. En esta sección veremos cómo invocar a estas funciones utilizando *eSQL* y *Entity Client*, algo que le parecerá igualmente muy similar a como estábamos acostumbrados a hacerlo hasta la llegada de EF.

Con el fin de mostrar un ejemplo práctico, partiremos del modelo creado en el capítulo anterior a lo largo de la sección "Trabajando con procedimientos almacenados", en el que vimos cómo importar un procedimiento almacenado llamado `usp_GetProductById` como una función del modelo, que llamamos `GetProductById`.

Una vez importada la función, simplemente deberemos establecer de forma correcta las propiedades `CommandText` y `CommandType` de nuestro objeto `EntityCommand`. La propiedad `CommandText` debe de establecerse especificando el nombre de la función de importación que deseamos invocar; en nuestro caso, ese nombre es `MamazonEntities.GetProductById`, donde `MamazonEntities` es el nombre del contenedor de entidades y `GetProductById` el nombre que hemos dado a la función de importación. Por supuesto, si la llamada al método necesita de algún parámetro, éstos deben establecerse de igual manera a como lo hicimos en la sección sobre consultas parametrizadas. El código de ejemplo de llamada a la función anterior puede verse en el siguiente listado.

```
// Creamos un objeto commando
EntityCommand command = connection.CreateCommand();
command.CommandText = "MamazonEntities.GetProductById";
command.CommandType = CommandType.StoredProcedure;

EntityParameter idProductoParameter =
   new EntityParameter("idProducto", DbType.Int32);
idProductoParameter.Value = 1000;
idProductoParameter.Direction = ParameterDirection.Input;

command.Parameters.Add(idProductoParameter);
```

1.2.5.- Revisión del código generado

En algunas ocasiones, puede interesarnos ver y/o almacenar el código *SQL* para el almacén concreto generado a partir de nuestras consultas en *eSQL*. Para ello, el objeto `EntityCommand` nos ofrece un método llamado `ToTraceString` que nos permitirá revisar de una forma realmente sencilla cuál es el código, en el dialecto de la base de datos subyacente, que se ejecutará contra la base de datos configurada.

```
public sealed class EntityCommand : DbCommand
{
   // ...
   public string ToTraceString();
}
```

Si al trozo de código de los listados anteriores añadimos la siguiente línea, podremos ver la sentencia *SQL* generada y ejecutada contra el motor de la base de datos, en nuestro caso *SQL Server*.

```
Console.WriteLine("SQL generado:{0}",
   command.ToTraceString());
```

1.2.6.- Transaccionabilidad

Existen a priori varias formas de manejar la transaccionalidad al trabajar con Entity Client. La primera de ellas, y a la postre la más cómoda, se basa en el uso de la clase `TransactionScope`, introducida en *.NET 2.0* con la librería `System.Transactions`. En este libro no vamos a abordar el uso de `TransactionScope`, algo que a estas alturas cualquier programador de *ADO.NET* seguramente conoce de sobra, aunque sí mostraremos, a continuación, un ejemplo sencillo en el que se hace uso de esta clase durante el trabajo con *Entity Client*.

```
using (TransactionScope scope =
   new TransactionScope(TransactionScopeOption.RequiresNew))
{
   connection.Open();

   // Creamos un objeto comando
   EntityCommand command = connection.CreateCommand();
   command.CommandText = "[Comando]";
   command.CommandType = CommandType.Text;

   // Obtenemos el  lector de la consulta
   EntityDataReader reader = command.ExecuteReader(
      CommandBehavior.SequentialAccess);

   VisitReader(reader as DbDataReader);

   scope.Complete();
   command.Dispose();
   reader.Close();
}
```

Recuerde la capacidad de promoción de transacciones que nos ofrece `TransactionScope`, siendo posible incluir dentro de su bloque el uso de distintos recursos transaccionales como otras bases de datos, mensajería asíncrona con ***MSMQ***, flujos de transacciones de ***WCF***, entre otros, y que el uso de esta capacidad lleva aparejado el requerimiento de que el servicio **MSDTC** (Coordinador de Transacciones Distribuídas) esté en funcionamiento.

Podríamos decir que el uso de la clase `TransactionScope` conlleva un manejo implícito de la transaccionalidad, pero, al igual que ocurre con el resto de proveedores de *ADO.NET*, que nos proporcionan clases herederas de `DbTransaction`, Entity Client también nos permite manejar la transaccionalidad de forma explícita mediante el uso de la clase `EntityTransaction`, cuyo funcionamiento es idéntico al ya conocido de *ADO.NET*. El proceso de manejar la transaccionalidad de forma explícita se basa en crear una nueva transacción a partir del objeto conexión, asignársela al comando o comandos que queramos que participen en la transacción y confirmar o deshacer la misma. En el siguiente listado vemos el proceso de manejo de una transacción de forma explícita en *Entity Client*.

```
connection.Open();

// Creamos una nueva transacción
EntityTransaction transaction =
   connection.BeginTransaction(
      System.Data.IsolationLevel.Serializable);

// Creamos un objeto comando
EntityCommand command = connection.CreateCommand();
command.CommandText = "[comando]";
command.CommandType = CommandType.Text;

// Le asignamos la transaccion
command.Transaction = transaction;

// Obtenemos el  lector de la consulta
EntityDataReader reader =
   command.ExecuteReader(CommandBehavior.SequentialAccess);

VisitReader(reader as DbDataReader);

transaction.Commit();// Confirmamos la transaccion
command.Dispose();
reader.Close();
```

1.2.7.- Cache de planes de consulta

Al igual que un motor de base de datos usa cachés para almacenar los distintos planes de ejecución, *Entity Client* también nos ofrece una caché de nuestras consultas *eSQL*. Esto nos abre una vía importante para mejorar el rendimiento de nuestras aplicaciones, puesto que las consultas *eSQL* compiladas y almacenadas en la caché no tendrán que volver a analizarse para generar el correspondiente dialecto *SQL* de la base

de datos con la que estemos trabajando. Para habilitar la posibilidad de que una consulta sea almacenada en la caché de planes de ejecución, únicamente debemos establecer a `true` el valor de la propiedad `EnablePlanCaching` del objeto `EntityCommand`.

```
EntityCommand command = connection.CreateCommand();
command.EnablePlanCaching = true;
```

Para verificar si una consulta está en la caché, *Entity Client* examina la consulta por su texto (**OJO**: teniendo en cuenta las diferencias entre mayúsculas y minúsculas), el número de parámetros y el tipo de los mismos. Ello hace altamente recomendable el uso de consultas parametrizadas y no el de consultas generadas dinámicamente, mediante la concatenación de cadenas de texto. Además, le recomendamos que no use los siguientes patrones de escritura de consultas, que consumen espacio dentro de la caché de planes de ejecución y pueden provocar que sentencias habituales salgan de la caché por falta de espacio:

- Cambios entre mayúsculas/minúsculas o dentro de las consultas.
- Agregar o eliminar espacios en blanco.
- Cambios en los valores de los literales de texto.
- Cambios en el texto de los comentarios.

CAPÍTULO

ObjectServices y Linq To Entities

1.- INTRODUCCIÓN

Si hace un breve repaso de lo visto hasta ahora en las páginas de este libro, puede resultarle chocante la diferencia entre lo presentado en el primer capítulo en relación con los modelos de dominio de los que tanto hemos hablado y la forma que tenemos de consultar los datos con *Entity Client*. Fíjese en que a pesar de obtener bastantes ventajas, como la posibilidad de trabajar con distintas fuentes de almacenamiento o la creación de modelos conceptuales, el proceso de lectura o consulta de datos es similar al que teníamos con *ADO.NET*, incluyendo todas sus deficiencias; incluso hasta se podría decir que es, en algunos aspectos, algo más tedioso, debido a la potencia de consulta que nos ofrece *eSQL*.

El principal de los problemas durante la consulta de datos con *Entity Client* tiene que ver con la materialización (generación de objetos fuertemente tipados a partir de los resultados). Como hemos visto en un listado del capítulo anterior, debemos ir analizando el tipo de datos que estamos leyendo hasta llegar a los tipos primitivos o complejos con los que construir nuestras clases de transporte de datos. Aquí es, sin duda alguna, donde podríamos tener una merma de productividad significativa, puesto que muchos de nuestros esfuerzos de codificación se centrarían en este aspecto, apartándonos de las reglas de negocio, que son las que verdaderamente dan respuesta a los problemas a resolver. Durante las siguientes secciones veremos cómo aprovechar los servicios de objetos que *EF* pone a nuestra disposición para simplificar sobremanera la materialización de objetos y la consulta del modelo conceptual.

2.- LOS SERVICIOS DE OBJETOS

Los servicios de objetos (*Object Services*) forman parte del conjunto de componentes que *EF* pone a nuestra disposición. Este componente nos da la

posibilidad de trabajar de un modo fuertemente tipado contra nuestros modelos de *EDM*, permitiendo consultar y manipular las entidades que conforman los mismos. Los tipos introducidos por los servicios de objetos están distribuidos por los espacios de nombres `System.Data.Objects` y `System.Data.Objects.DataClasses`.

Los principales elementos que forman parte de los servicios de objetos son las entidades y el contexto de trabajo, elementos que presentaremos a continuación.

2.1.- El contexto de trabajo

El **contexto de trabajo** o **contexto de objetos** es la pieza fundamental que nos permitirá trabajar con los objetos, tanto para su consulta como para la posterior actualización (inserción, modificación y borrado) de los mismos. Un contexto de trabajo, en realidad, no es más que la implementación del patrón **Unit Of Work**, definido por **Martin Fowler** en su más que conocida obra **PoEAA**. Por lo tanto, según este patrón en realidad la responsabilidad de un contexto de trabajo será la de *mantener una lista de objetos afectados en una transacción y coordinar la escritura de los cambios así como la resolución de problemas de concurrencia.*

En ADO.NET Entity Framework los contextos de trabajo vienen representados por la clase `ObjectContext`, definida dentro del espacio de nombres `System.Data.Objects`, la estructura básica de este objeto la podemos ver a continuación:

```
public class ObjectContext : IDisposable
{
  public ObjectContext(EntityConnection connection);

  public ObjectContextOptions ContextOptions { get; }

  public MetadataWorkspace MetadataWorkspace { get; }

  public ObjectStateManager ObjectStateManager { get; }

  public void AddObject(string entitySetName, object entity);

  public ObjectSet<TEntity> CreateObjectSet<TEntity>()
  where TEntity : class;

  public void DeleteObject(object entity);

  public void Detach(object entity);

  public int SaveChanges();

  //Abreviada para facilitar la lectura
}
```

Además de ser una unidad de trabajo (UoW) los contextos de trabajo definidos por ADO.NET Entity Framework tienen otras responsabilidades:

- El contexto de trabajo contiene la información del modelo contra el que trabaja; o sea, toda la información de metadatos en sus tres secciones, SSDL,

CSDL y MSL. Esta información se expone al mundo exterior mediante la propiedad `MetadataWorkspace`, que devuelve un tipo del mismo nombre.

- El contexto de trabajo es el encargado de manejar las conexiones con el modelo EDM, y para ello se encarga de almacenar y manejar un objeto de tipo `EntityConnection`.
- El contexto es el encargado de realizar un seguimiento sobre las entidades consultadas y el seguimiento de sus cambios mediante la propiedad `ObjectStateManager`, que devuelve un tipo del mismo nombre.

Además de lo anterior, para nosotros un objeto `ObjectContext` será la base principal para cualquier tarea que deseemos realizar sobre el modelo conceptual, desde la consulta hasta la manipulación de elementos: fíjese como dentro de la estructura de un contexto de trabajo se incluyen los métodos `AddObject` y `DeleteObject`.

Nota: En la primera versión del producto, la agregación y el borrado eran elementos de obligatorio uso dentro de un contexto, cuando no debería ser así, o por lo menos no solamente como métodos del contexto. En esta nueva versión del producto, EF 4.1, tanto los métodos de agregación como de borrado están también disponibles en un nuevo elemento **IObjectSet**, del cual hablaremos más adelante, gracias al cual podremos hacer implementaciones de nuestros repositorios más correctamente.

Cada vez que un nuevo modelo *EDM* es creado mediante el asistente integrado dentro de *Visual Studio 2010*, además del correspondiente modelo conceptual se generan de forma automática una serie de clases. Una de estas clases generadas se conoce como el ***contenedor de entidades*** o `EntityContainer`. Recuerde que un contenedor de entidades no es más que una agrupación lógica de elementos `EntitySet` y/o `AssociationSet`. Ese contenedor de entidades se nos presenta en código como una clase que hereda de `ObjectContext` y que va a encapsular todo lo necesario para el trabajo con nuestros modelos de una forma fuertemente tipada.

Suponga que creamos un nuevo modelo de entidades, **code\EF4_EX_16**, que únicamente contenga una entidad *Customer* como cualquiera de las definidas en el segundo capítulo. Si revisamos el código generado a partir de este modelo, nos encontraremos con una estructura de nuestro contexto de trabajo similar a la que se presenta en el siguiente listado.

```
public partial class EF4EXEntities : ObjectContext
{
    #region Constructors

/// <summary>
/// Initializes a new EF4EXEntities object using the connection
string found in the 'EF4EXEntities' section of the application
configuration file.
    /// </summary>
```

```
        public EF4EXEntities() : base("name=EF4EXEntities",
"EF4EXEntities")
        {
            this.ContextOptions.LazyLoadingEnabled = true;
            OnContextCreated();
        }

        /// <summary>
        /// Initialize a new EF4EXEntities object.
        /// </summary>
        public EF4EXEntities(string connectionString) :
base(connectionString, "EF4EXEntities")
        {
            this.ContextOptions.LazyLoadingEnabled = true;
            OnContextCreated();
        }

        /// <summary>
        /// Initialize a new EF4EXEntities object.
        /// </summary>
        public EF4EXEntities(EntityConnection connection) :
base(connection, "EF4EXEntities")
        {
            this.ContextOptions.LazyLoadingEnabled = true;
            OnContextCreated();
        }

        #endregion

        #region Partial Methods

        partial void OnContextCreated();

        #endregion

        #region ObjectSet Properties

        /// <summary>
        /// No Metadata Documentation available.
        /// </summary>
        public ObjectSet<Customer> Customers
        {
            get
            {
                if ((_Customers == null))
                {
                    _Customers =
base.CreateObjectSet<Customer>("Customers");
                }
                return _Customers;
            }
        }
        private ObjectSet<Customer> _Customers;

        #endregion

        #region AddTo Methods

        /// <summary>
        /// Deprecated Method for adding a new object to the
Customers EntitySet. Consider using the .Add method of the associated
```

```
ObjectSet&lt;T&gt; property instead.
        /// </summary>
        public void AddToCustomers(Customer customer)
        {
            base.AddObject("Customers", customer);
        }

        #endregion
}
```

Si observa detenidamente el código del contexto de trabajo generado, podrá ver como efectivamente esta clase hereda de `ObjectContext` (disponiendo por tanto de toda la funcionalidad que ésta nos da), y la extiende con recursos personalizados para el modelo, de forma similar a como un conjunto de datos tipado de ADO.NET extiende a la clase `DataSet`. Además, nuestro **contexto tipado** nos proporciona tres sobrecargas diferentes de su constructor. La primera de ellas, el constructor por defecto, hace uso del nombre dado al contenedor y del nombre de la cadena de conexión a utilizar, que deberemos tener en el fichero de configuración del ejecutable que haga uso del modelo.

Las otras dos sobrecargas del constructor nos permitirán especificar directamente la cadena de conexión de *Entity Client* a utilizar, o el objeto `EntityConnection` que se utilizará para aplicar las distintas operaciones sobre la base de datos subyacente.

Además de las sobrecargas del constructor, esta clase nos ofrece, para cada una de los `EntitySet` contenidos en el modelo EDM, un método llamado `Add[NombreEntitySet]` y una propiedad de igual nombre que el `EntitySet` que nos devuelve un objeto `ObjectSet<T>`, del que hablaremos posteriormente ya que es una pieza clave dentro de los servicios de objetos.

Dentro de las nuevas funcionalidades incluidas en ADO.NET Entity Framework 4.1 podemos encontrar algunos elementos en la definición de nuestros contextos de trabajo que tienen por objetivo simplificarnos el trabajo y aportarnos algunas posibles soluciones ante determinados escenarios. Entre estos elementos, podemos encontrar por ejemplo nuevos métodos que nos permitirán ejecutar tanto consultas como comandos *adhoc* contra el motor de base de datos subyacente que tengamos.

En el siguiente fragmento puede ver la firma de estos nuevos métodos:

```
public class ObjectContext : IDisposable
{
        public int ExecuteStoreCommand(string commandText, params
object[] parameters);

        public ObjectResult<TElement>
ExecuteStoreQuery<TElement>(string commandText, params object[]
parameters);

        public ObjectResult<TEntity>
ExecuteStoreQuery<TEntity>(string commandText, string entitySetName,
MergeOption mergeOption, params object[] parameters);

    // ...
}
```

Dependiendo de su orientación como desarrollador y de la madurez que tenga en cuanto al trabajo con modelos de dominio, estos métodos podrán parecerle una pequeña aberración. La verdad es que, aunque incluir consultas *adhoc* puede romper algunos de los principios sobre los que estamos hablando desde el primer capítulo, no es menos cierto que negar la posible existencia de puntos críticos que necesiten de este grado de especialidad es igualmente negativo. El uso de estos métodos es sencillo. Para verlo partiremos de un modelo muy simple, **code\ef4_ex_17,** con una única entidad *Customer* tal y como se ve en la siguiente imagen:

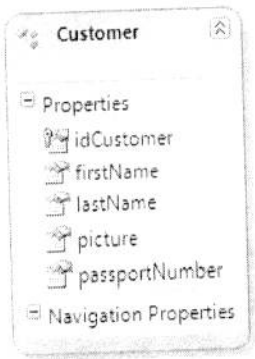


Figura 4.1.- ***Entidad customer dentro de un modelo de entidades***

Una vez generado el modelo, tal y como se acaba de explicar, automáticamente disponemos de una serie de clases en el archivo de diseñador asociado. Entre estas clases, en nuestro ejemplo, disponemos de *EF4_EXEntities*, nuestro contexto de entidades.

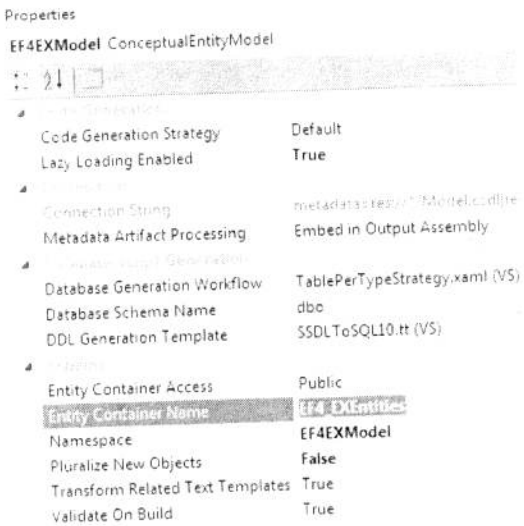


Figura 4.2.- Modificación del nombre del contexto de entidades

Nota: Aunque el nombre de este contexto se establece en el asistente de creación, el mismo puede ser modificado desde su ventana de propiedades, tal y como se muestra en la figura 4.2. Puesto que solamente perseguimos fines didácticos no modificaremos estos nombres, contexto y/o entidades, ya que este trabajo solamente introduciría un ruido que no aportaría nada en este momento.

Empezaremos mostrando, en el siguiente fragmento de código, el uso de `ExecuteStoreQuery` para ejecutar una consulta de forma directa contra la base de datos subyacente y mapear los resultados a una entidad `Customer`.

```
using (EF4_EXEntities context = new EF4_EXEntities())
{
    string sql = "SELECT
idCustomer,firstName,lastName,picture,passportNumber" ;
    sql += "FROM dbo.Customer WHERE idCustomer > {0}";

    int customerId = 1;

    ObjectResult<Customer> customers =
context.ExecuteStoreQuery<Customer>(sql,customerId);
}
```

En este último fragmento, `ExecuteStoreQuery`, realiza la consulta establecida en la variable *sql*, consulta parametrizada, aunque no como estamos acostumbrados en *SqlClient* por ejemplo, por medio del uso de los segmentos {index}, dónde *index* indica el orden de cada uno de los elementos del parámetro *parameters* de este método.

`ExecuteStoreCommand`, tiene un funcionamiento prácticamente igual aunque en este caso permite ejecutar comandos contra la base de datos, una actualización, un borrado etc. En el siguiente fragmento mostraremos un ejemplo de `ExecuteStoreCommand` para actualizar un elemento.

```
using (EF4_EXEntities context = new EF4_EXEntities())
{
      string sql = "UPDATE Customer SET firstName={0} WHERE
idCustomer={1}";

      int customerId = 1;
      string firstName = "Unai";

      int rowsAffected =
context.ExecuteStoreCommand(sql,firstName,customerId);
}
```

Nota: Aunque no se ha hablado de ellos, nuestros contextos de trabajo (ObjectContext), disponen de métodos para comprobar la existencia y autogenerar una base de datos igual que la disponible en las definiciones incluidas en nuestro SSDL. Estos nuevos métodos incluidos en .NET Entity Framework 4.1 son:

- **DatabaseExists** para comprobar la existencia de la base de datos definida en nuestro archivo de configuración.
- **CreateDatabase** para crear la base de datos a partir de la definición de los elementos incluidos en el espacio SSDL.
- **CreateDatabaseScript** para generar el script de creación

2.2.- Las entidades

Junto con la creación del contexto de trabajo, el diseñador de EDM también se encarga de generar (a través de la herramienta **EdmGen.exe** que presentamos en el capítulo de introducción) una clase por cada una de las entidades (`EntityType`) del modelo conceptual; en este caso, una para cada `EntityType`, y no solamente para los `EntitySet` como ocurre en el caso de las propiedades de tipo `ObjectSet<T>` del contexto. Cada una de estas clases representa a una entidad del modelo conceptual, y nos permitirán trabajar de una forma tipada, puesto que haremos uso de instancias de esas clases y trabajaremos con sus distintas propiedades.

```
[EdmEntityTypeAttribute(NamespaceName="EF4EXModel", Name="Customer")]
    [Serializable()]
    [DataContractAttribute(IsReference=true)]
    public partial class Customer : EntityObject
    {
        #region Factory Method

        /// <summary>
        /// Create a new Customer object.
        /// </summary>
        /// <param name="idCustomer">Initial value of the idCustomer
property.</param>
        /// <param name="firstName">Initial value of the firstName
property.</param>
        /// <param name="lastName">Initial value of the lastName
property.</param>
        /// <param name="passportNumber">Initial value of the
passportNumber property.</param>
        public static Customer CreateCustomer(global::System.Int32
idCustomer, global::System.String firstName, global::System.String
lastName, global::System.String passportNumber)
```

```
        {
            Customer customer = new Customer();
            customer.idCustomer = idCustomer;
            customer.firstName = firstName;
            customer.lastName = lastName;
            customer.passportNumber = passportNumber;
            return customer;
        }

        #endregion
        #region Primitive Properties

        /// <summary>
        /// No Metadata Documentation available.
        /// </summary>
        [EdmScalarPropertyAttribute(EntityKeyProperty=true,
IsNullable=false)]
        [DataMemberAttribute()]
        public global::System.Int32 idCustomer
        {
            get
            {
                return _idCustomer;
            }
            set
            {
                if (_idCustomer != value)
                {
                    OnidCustomerChanging(value);
                    ReportPropertyChanging("idCustomer");
                    _idCustomer =
StructuralObject.SetValidValue(value);
                    ReportPropertyChanged("idCustomer");
                    OnidCustomerChanged();
                }
            }
        }
        private global::System.Int32 _idCustomer;
        partial void OnidCustomerChanging(global::System.Int32
value);
        partial void OnidCustomerChanged();

        /// <summary>
        /// No Metadata Documentation available.
        /// </summary>
        [EdmScalarPropertyAttribute(EntityKeyProperty=false,
IsNullable=false)]
        [DataMemberAttribute()]
        public global::System.String firstName
        {
            get
            {
                return _firstName;
            }
            set
            {
                OnfirstNameChanging(value);
                ReportPropertyChanging("firstName");
                _firstName = StructuralObject.SetValidValue(value,
false);
                ReportPropertyChanged("firstName");
```

```
                OnfirstNameChanged();
            }
        }
        private global::System.String _firstName;
        partial void OnfirstNameChanging(global::System.String
value);
        partial void OnfirstNameChanged();

        /// <summary>
        /// No Metadata Documentation available.
        /// </summary>
        [EdmScalarPropertyAttribute(EntityKeyProperty=false,
IsNullable=false)]
        [DataMemberAttribute()]
        public global::System.String lastName
        {
            get
            {
                return _lastName;
            }
            set
            {
                OnlastNameChanging(value);
                ReportPropertyChanging("lastName");
                _lastName = StructuralObject.SetValidValue(value,
false);
                ReportPropertyChanged("lastName");
                OnlastNameChanged();
            }
        }
        private global::System.String _lastName;
        partial void OnlastNameChanging(global::System.String value);
        partial void OnlastNameChanged();

        /// <summary>
        /// No Metadata Documentation available.
        /// </summary>
        [EdmScalarPropertyAttribute(EntityKeyProperty=false,
IsNullable=true)]
        [DataMemberAttribute()]
        public global::System.Byte[] picture
        {
            get
            {
                return StructuralObject.GetValidValue(_picture);
            }
            set
            {
                OnpictureChanging(value);
                ReportPropertyChanging("picture");
                _picture = StructuralObject.SetValidValue(value,
true);
                ReportPropertyChanged("picture");
                OnpictureChanged();
            }
        }
        private global::System.Byte[] _picture;
        partial void OnpictureChanging(global::System.Byte[] value);
        partial void OnpictureChanged();

        /// <summary>
```

```
        /// No Metadata Documentation available.
        /// </summary>
        [EdmScalarPropertyAttribute(EntityKeyProperty=false,
IsNullable=false)]
        [DataMemberAttribute()]
        public global::System.String passportNumber
        {
            get
            {
                return _passportNumber;
            }
            set
            {
                OnpassportNumberChanging(value);
                ReportPropertyChanging("passportNumber");
                _passportNumber =
StructuralObject.SetValidValue(value, false);
                ReportPropertyChanged("passportNumber");
                OnpassportNumberChanged();
            }
        }
        private global::System.String _passportNumber;
        partial void OnpassportNumberChanging(global::System.String
value);
        partial void OnpassportNumberChanged();

        #endregion

    }
```

Si revisa el código del fragmento anterior para nuestra entidad `Customer` del modelo creado anteriormente, podrá observar que ésta hereda de una clase base llamada `EntityObject`, definida en `System.Data.Objects.DataClasses`. Este tipo base es el que contiene la funcionalidad básica que permitirá que nuestras entidades sean seguidas por los contextos de trabajo, enterándose éste último de cuándo se realizan cambios sobre ellas. Adicionalmente, los `EntityObject` almacenan una clave (un objeto de tipo `EntityKey`), y pueden contener relaciones de cualquier cardinalidad con otras entidades del modelo.

Nota: En nuestra literatura las clases que heredan de una clase base se conocen habitualmente como 'Clases Prescriptivas'.

El hecho de disponer o no de *clases prescriptivas* como las clases que representan a nuestro dominio no es algo sobre lo que debamos pasar de puntillas, puesto que, este hecho marca de forma importantísima nuestros diseños de soluciones. Cuando uno se pone a trabajar en una solución guiada por un modelo de dominio, y en general en cualquier solución, siempre trata de mantener principios básicos como el desacoplamiento y la separación de responsabilidades. En modelos de dominio, nuestras entidades y los posibles *Value Objects* extraídos con ellas deberían concentrarse solamente en el *negocio* para el que fueron definidas. No deberían tener que conocer nada acerca de la persistencia de las mismas. Si, como hemos visto en este ejemplo anterior, nuestra entidad hereda de *EntityObject*, en el fondo está imponiendo

un conocimiento -exagerado además- de la persistencia, puesto que esta clase como hemos visto está definida dentro de namespaces y librerías propias de Entity Framework, amén de otras implicaciones.

Nota: Cuando se habla del concepto de *ignorancia de la persistencia* se suele tender a pensar que este principio está pensado con la finalidad de facilitar la sustitución de un motor por otro: Entity Framework por Hibernate por ejemplo. En realidad, este principio no está pensado para esto, que pocas veces se da en la realidad de una solución, sino para facilitar conceptos como el desacoplamiento y la separación de responsabilidades entre otros.

Perseguir el principio de *ignorancia de la persistencia,* ha sido seguramente una de las principales prioridades del grupo de trabajo en la realización de esta nueva versión de ADO.NET Entity Framework. De este trabajo, salen nuevas capacidades como la posibilidad de *customizar* cómo queremos que se generen estas entidades. Aunque por defecto EDM nos genere clases prescriptivas como las que acabamos de ver (la única opción en la versión anterior del producto), con unos sencillos pasos podemos modificar esta generación. Para ello nos valdremos de los artefactos de generación que tengamos disponibles. Para mostrar estos artefactos, **code\ef4_ex_18**, solamente tenemos que seleccionar la opción del menú contextual del modelo *"Add Code Generation Item o Agregar artefacto de generación de código"*, figura 4.3.

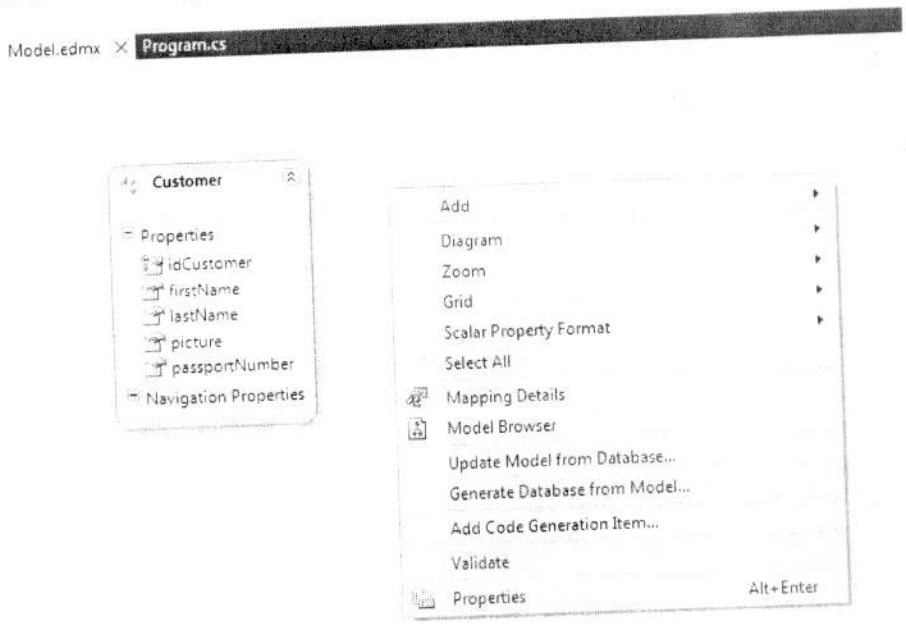


Figura 4.3.- Selección del artefacto de generación

Una vez seleccionada esta opción, *Visual Studio 2010* nos mostrará los distintos artefactos que tenemos disponibles, Figura 4.4. Por defecto en Visual Studio solamente disponemos de dos artefactos incluidos: el generador de clases prescriptivas, ya visto, y el generador de un tipo de entidades conocidas como **Selft Tracking Entities**, más adelante STE. Adicionalmente, disponemos de extensiones, creadas por el propio grupo de producto y no incluidas en la *release* de VS 2010 por falta de tiempo, para la generación de objetos **POCO**, Figura 4.5.

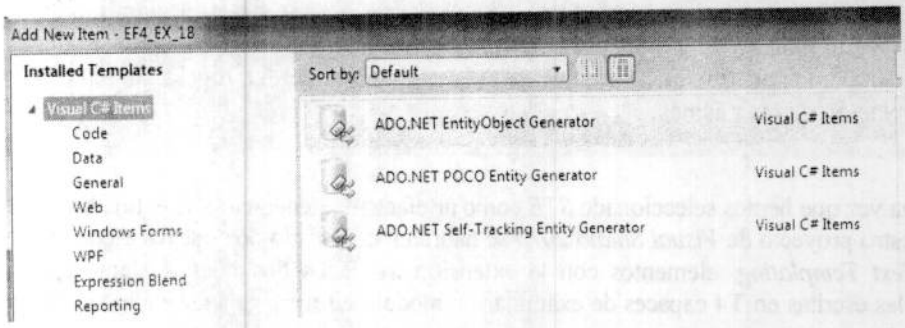


Figura 4.4.- Selección de artefactos de generación

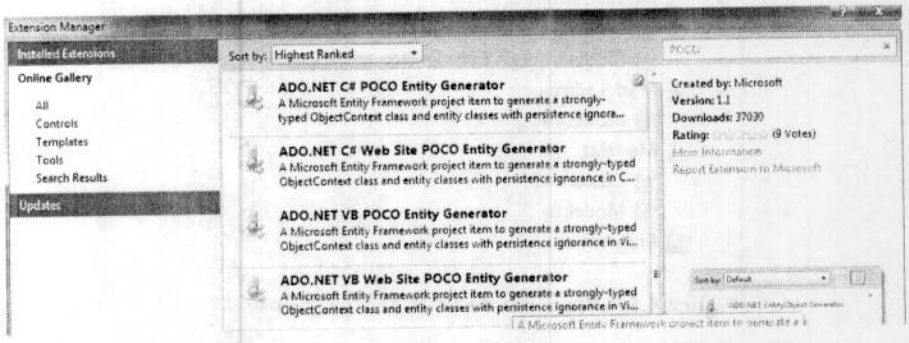


Figura 4.5.- Artefacto de generación de entidades POCO

Nota: Para acceder a estas nuevas extensiones de Visual Studio 2010 así como otras muchas puede utilizar **Extension Manager**, herramienta integrada dentro del menú herramientas que nos permite acceder a las distintas extensiones que tenemos disponibles, tanto gratuitas como no.

2.2.1.- Selft Tracking Entities

Si mantener "el principio de ignorancia de la persistencia" era una de las prioridades del grupo de trabajo, mejorar las capacidades de trabajo en arquitecturas en N-Capas era sin duda otra, sobre todo teniendo en cuenta la cantidad de *feedback* recibido desde las distintas

comunidades de *.NET*. Podríamos definir a las entidades STE (recuerde: *Selft Tracking Entities*) como una solución de compromiso de estas dos prioridades. Por un lado, las entidades *STE* son ignorantes de la persistencia, en tanto en cuanto no tienen y no necesitan conocimiento alguno de *ADO.NET Entity Framework*, ni por supuesto, referencias a sus librerías. Y por otro lado facilitan sin medida el trabajo en *Arquitecturas N-Capas* como iremos viendo a lo largo de este y el siguiente capítulo principalmente.

Nota: Este último párrafo daría realmente mucho para discutir. Efectivamente el código de una entidad STE no necesita referenciar ninguna librería de ADO.NET puesto que toda su infraestructura se basa en código que podemos asumir como nuestro. Sin embargo, todo este código, aunque no de forma explícita, se podría decir que está *abocado* a trabajar con el sistema de tracking de Entity Framework.

El código generado por los artefactos STE y POCO se basa en plantillas de Text Templating y el lenguaje T4, en un apéndice de este libro dispone de una pequeña guía de referencia que le permitirá comprender el funcionamiento y adaptación de las mismas.

Una vez que hemos seleccionado *STE* como artefacto de generación de entidades dentro de nuestro proyecto de *Visual Studio 2010* se habrán incorporado dos nuevos elementos de tipo *Text Templating*, elementos con la extensión **.tt**. Estos dos nuevos elementos son plantillas escritas en **T4** capaces de examinar un modelo **edmx** y generar código en base a él.

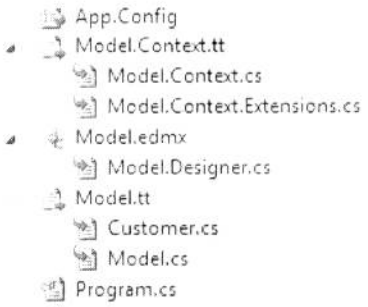


Figura 4.6.- Plantillas de Selft Tracking Entities

Estos dos nuevos elementos se encargarán de la generación de las entidades del modelo, *Model.tt*, y del código de nuestro contexto de trabajo, *Model.Context.tt*. Piense que esta separación es lógica, puesto que en un solución real las entidades y el contexto estarán separadas físicamente, por lo tanto, disponer de dos plantillas diferenciadas es un ahorro importante al comienzo de un nuevo proyecto o modulo. Ahora, con STE, las entidades de nuestro modelo ya no necesitarán ser subclases de `EntityObject`.

Fíjese en el siguiente fragmento de la clase `Customer` generada con este nuevo artefacto:

```
[DataContract(IsReference = true)]
    public partial class Customer: IObjectWithChangeTracker,
INotifyPropertyChanged
    {
        #region Primitive Properties

        [DataMember]
        public int idCustomer
        {
            get { return _idCustomer; }
            set
            {
                if (_idCustomer != value)
                {
                    if (ChangeTracker.ChangeTrackingEnabled &&
ChangeTracker.State != ObjectState.Added)
                    {
                        throw new InvalidOperationException("The
property 'idCustomer' is part of the object's key and cannot be
changed. Changes to key properties can only be made when the object
is not being tracked or is in the Added state.");
                    }
                    _idCustomer = value;
                    OnPropertyChanged("idCustomer");
                }
            }
        }
        private int _idCustomer;

        [DataMember]
        public string firstName
        {
            get { return _firstName; }
            set
            {
                if (_firstName != value)
                {
                    _firstName = value;
                    OnPropertyChanged("firstName");
                }
            }
        }
        private string _firstName;

        [DataMember]
        public string lastName
        {
            get { return _lastName; }
            set
            {
                if (_lastName != value)
                {
                    _lastName = value;
                    OnPropertyChanged("lastName");
                }
            }
        }
        private string _lastName;
```

```
        [DataMember]
        public byte[] picture
        {
            get { return _picture; }
            set
            {
                if (_picture != value)
                {
                    _picture = value;
                    OnPropertyChanged("picture");
                }
            }
        }
        private byte[] _picture;

        [DataMember]
        public string passportNumber
        {
            get { return _passportNumber; }
            set
            {
                if (_passportNumber != value)
                {
                    _passportNumber = value;
                    OnPropertyChanged("passportNumber");
                }
            }
        }
        private string _passportNumber;

        #endregion
        #region ChangeTracking

        protected virtual void OnPropertyChanged(String propertyName)
        {
            if (ChangeTracker.State != ObjectState.Added &&
ChangeTracker.State != ObjectState.Deleted)
            {
                ChangeTracker.State = ObjectState.Modified;
            }
            if (_propertyChanged != null)
            {
                _propertyChanged(this, new
PropertyChangedEventArgs(propertyName));
            }
        }

        protected virtual void OnNavigationPropertyChanged(String
propertyName)
        {
            if (_propertyChanged != null)
            {
                _propertyChanged(this, new
PropertyChangedEventArgs(propertyName));
            }
        }

        event PropertyChangedEventHandler
INotifyPropertyChanged.PropertyChanged{ add { _propertyChanged +=
value; } remove { _propertyChanged -= value; } }
        private event PropertyChangedEventHandler _propertyChanged;
```

```
        private ObjectChangeTracker _changeTracker;

        [DataMember]
        public ObjectChangeTracker ChangeTracker
        {
            get
            {
                if (_changeTracker == null)
                {
                    _changeTracker = new ObjectChangeTracker();
                    _changeTracker.ObjectStateChanging +=
HandleObjectStateChanging;
                }
                return _changeTracker;
            }
            set
            {
                if(_changeTracker != null)
                {
                    _changeTracker.ObjectStateChanging -=
HandleObjectStateChanging;
                }
                _changeTracker = value;
                if(_changeTracker != null)
                {
                    _changeTracker.ObjectStateChanging +=
HandleObjectStateChanging;
                }
            }
        }

        private void HandleObjectStateChanging(object sender,
ObjectStateChangingEventArgs e)
        {
            if (e.NewState == ObjectState.Deleted)
            {
                ClearNavigationProperties();
            }
        }

        protected bool IsDeserializing { get; private set; }

        [OnDeserializing]
        public void OnDeserializingMethod(StreamingContext context)
        {
            IsDeserializing = true;
        }

        [OnDeserialized]
        public void OnDeserializedMethod(StreamingContext context)
        {
            IsDeserializing = false;
            ChangeTracker.ChangeTrackingEnabled = true;
        }

        protected virtual void ClearNavigationProperties()
        {
        }

        #endregion
    }
```

Como habrá observado de este fragmento, la entidad `Customer` no depende de ningún elemento de **ADO.NET Entity Framework,** solamente de dos interfaces `IObjectWithChangeTracker` y `INotifyPropertyChanged`. El segundo de estos contratos seguramente ya le resultará conocido si ha trabajado con **Windows Forms** o **Windows Presentation Foundation** puesto que es parte común para las notificaciones de enlace a datos o data binding, y para este propósito está. Por el contrario, la interfaz `IObjectWithChangeTracker` es un nuevo contrato que nos permitirá definir el sistema de gestión de cambios a usar por la entidad. En realidad también la podríamos definir como un adaptador de la información de cambios entre la entidad y el contexto de trabajo. Puesto que aún no sabemos nada acerca de cómo nuestros contextos realizan el seguimiento de las entidades por ahora no profundizaremos más en este tema y lo retomaremos más adelante.

La segunda de las plantillas incluidas en nuestro proyecto, con el artefacto de generación STE, realiza la tarea de generar el contexto de trabajo y una serie de métodos extensores que permiten trabajar con nuestras entidades y especialmente con los objetos de tipo `IObjectWithChangeTracker`. Cuando retomemos las entidades y especialmente el trabajo con esta última interfaz nos extenderemos en estos métodos extensores.

2.2.2.- Entidades POCO

Aunque las entidades STE se podrían categorizar como ***ignorantes de la persistencia***, admitiendo algo de discusión sobre este tema, éstas necesitan de un código de soporte, el definido por la interfaz `IObjectWithChangeTracker` y su método `ChangeTracker`.

```
public interface IObjectWithChangeTracker
{
        ObjectChangeTracker ChangeTracker { get; }
}
```

Aunque este código de soporte sea nuestro, en el sentido de que el mismo está generado por medio de las plantillas **.tt,** implica una restricción importante si decide exponer las entidades STE en una fachada de servicios, puesto que los clientes de estos servicios necesitarán este código y por lo tanto estamos restringiendo la tecnología de los mismos a clientes **.NET** solamente. Por supuesto, esto no es admisible en todos los escenarios, puesto que muchas veces nuestros servicios tienen a distintas tecnologías como clientes y por lo tanto la interoperabilidad es necesaria.

Nota: El hecho de exponer nuestras entidades fuera de un modelo de dominio no es en si la mejor práctica, puesto que tiene un impacto en los clientes si la frecuencia de cambio en el mismo es alta, además de, por supuesto, dar información del sentido dominio a clientes que no tendrían por qué tenerla. Admitiendo estos puntos, la decisión de exponer las entidades del dominio en una fachada de servicios suele estar provocada por razones de sencillez y productividad.

Por suerte, como comentamos, además del artefacto de generación de STE, disponemos dentro del *administrador de extensiones* de Visual Studio artefactos para la generación de objetos **POCO** o **Plain Old CLR Objects**. Este tipo de objetos, se suelen definir como aquellos que no heredan de ninguna clase base ni implementan ninguna interfaz, aunque esto no es más que una forma de asegurarnos que el principio de *ignorancia de la persistencia* está vivo. Es común que estas entidades puedan apoyarse en contratos del dominio (por ejemplo para validación) sin necesidad de que esto rompa este concepto. En el ejemplo **code\ef4_ex_19**, mostramos como partiendo del mismo modelo que en ejemplo de *Selft Tracking Entities* podemos seleccionar el artefacto de generación de objetos POCO, figura 4.7, el cual, al igual que en el caso anterior, incorporará a nuestro proyecto dos nuevas plantillas T4.

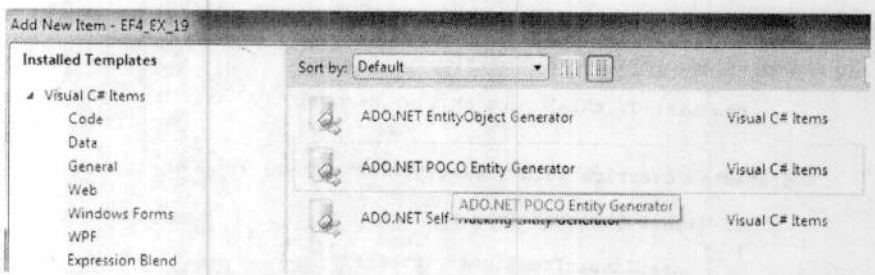


Figura 4.7.- Generación con el artefacto de plantillas POCO

Si examinamos los elementos generados por las nuevas plantillas podremos observar como el código del contexto de trabajo es idéntico al de nuestra plantilla STE. Lógicamente ahora no se dispone de métodos extensores para trabajar con `IObjectWithChangeTracker`. Sin embargo, podemos notar como ha cambiado notablemente nuestro código de entidades, puesto que las entidades generadas ni necesitan heredar de `EntityObject` ni implementar ninguna interfaz de soporte. A continuación podemos ver un fragmento de código con nuestra entidad Customer generada con el nuevo artefacto de generación.

```
public partial class Customer
{
        #region Primitive Properties

        public virtual int idCustomer
```

```
    {
        get;
        set;
    }

    public virtual string firstName
    {
        get;
        set;
    }

    public virtual string lastName
    {
        get;
        set;
    }

    public virtual byte[] picture
    {
        get;
        set;
    }

    public virtual string passportNumber
    {
        get;
        set;
    }

    #endregion
}
```

Seguramente, le habrá sorprendido que las propiedades de esta entidad estén generadas como virtuales. Realmente este hecho tiene que ver con una funcionalidad que trataremos, después de mostrar el mecanismo de gestión de cambios de los contextos de trabajo.

Además del código de las entidades, la plantilla de generación crea un nuevo elemento heredado de `ObservableCollection`:

```
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);
        }
    }
}
```

Este nuevo elemento, `FixupCollection`, servirá como tipo para las propiedades de navegación, colecciones, que tengamos incluidas dentro de nuestro modelo. El término *Fixup* viene por su uso para el mantenimiento de las relaciones, es decir, la capacidad de asegurarnos que las relaciones entre unas entidades y otras son coherentes. Para ver

esto en más detalle incorporaremos una nueva entidad a nuestro modelo, **sql\EF4_EX_14.sql**, por ejemplo `Orders`, y mostraremos nuevamente el código de la entidad `Customer`.

```
public partial class Customer
    {
        #region Primitive Properties

        public virtual int idCustomer
        {
            get;
            set;
        }

        public virtual string firstName
        {
            get;
            set;
        }

        public virtual string lastName
        {
            get;
            set;
        }

        public virtual byte[] picture
        {
            get;
            set;
        }

        public virtual string passportNumber
        {
            get;
            set;
        }

        #endregion
        #region Navigation Properties

        public virtual ICollection<Orders> Orders
        {
            get
            {
                if (_orders == null)
                {
                    var newCollection = new
FixupCollection<Orders>();
                    newCollection.CollectionChanged += FixupOrders;
                    _orders = newCollection;
                }
                return _orders;
            }
            set
            {
                if (!ReferenceEquals(_orders, value))
                {
                    var previousValue = _orders as
```

```
FixupCollection<Orders>;
                    if (previousValue != null)
                    {
                        previousValue.CollectionChanged -=
FixupOrders;
                    }
                    _orders = value;
                    var newValue = value as FixupCollection<Orders>;
                    if (newValue != null)
                    {
                        newValue.CollectionChanged += FixupOrders;
                    }
                }
            }
        }
        private ICollection<Orders> _orders;

        #endregion
        #region Association Fixup

        private void FixupOrders(object sender,
NotifyCollectionChangedEventArgs e)
        {
            if (e.NewItems != null)
            {
                foreach (Orders item in e.NewItems)
                {
                    item.Customer = this;
                }
            }

            if (e.OldItems != null)
            {
                foreach (Orders item in e.OldItems)
                {
                    if (ReferenceEquals(item.Customer, this))
                    {
                        item.Customer = null;
                    }
                }
            }
        }

        #endregion
    }
```

Como podrá observar, en nuestra propiedad de navegación `Orders`, se obtiene una colección de tipo `Fixup`, colección en la cual se agrega un manejador para su evento `CollectionChanged`.

```
public virtual ICollection<Orders> Orders
{
            get
            {
                if (_orders == null)
                {
                    var newCollection = new
FixupCollection<Orders>();
```

```
                newCollection.CollectionChanged += FixupOrders;
                _orders = newCollection;
            }
            return _orders;
        }
        set
        {
          //...
        }
    }
```

El objetivo de este manejador será el de mantener la coherencia entre los pedidos de este cliente, y el valor de la propiedad `Cliente` de los mismos. Recuerde que cuando hay una navegación disponemos de propiedades de navegación en ambos sentidos.

```
private void FixupOrders(object sender,
NotifyCollectionChangedEventArgs e)
{
        if (e.NewItems != null)
        {
            foreach (Orders item in e.NewItems)
            {
                item.Customer = this;
            }
        }

        if (e.OldItems != null)
        {
            foreach (Orders item in e.OldItems)
            {
                if (ReferenceEquals(item.Customer, this))
                {
                    item.Customer = null;
                }
            }
        }
}
```

2.3.- ObjectSet<TEntity>

Ya hemos visto qué es un contexto de trabajo y cómo es el código que genera automáticamente el asistente de modelos EDM. También hemos visto cómo este mismo asistente, o bien las plantillas **T4** si hemos seleccionado algún artefacto de generación, genera nuevas clases para cada una de las entidades y las características y capacidades de éstas. Pero aún no sabemos nada acerca de cómo todo esto puede mejorar nuestro trabajo con los modelos con relación a lo visto en el capítulo anterior sobre Entity Client. Es precisamente en este punto donde entra en juego el tipo `ObjectSet<TEntity>`, puesto que esta clase representa básicamente una consulta que devuelve una colección de cero o más tipos de entidades. Cada objeto de este tipo está ligado a un contexto de trabajo que contiene la conexión y la información de metadatos necesarios para realizar la consulta y materializar los resultados.

Para ser francos, `ObjectSet<TEntity>` no es más que un envoltorio sobre `ObjectQuery<TEntity>`, tipo que realmente realiza las funciones anteriormente mencionadas. La existencia de este *wrapper*, nuevo en *Entity Framework 4.1*, nos permite agregar una serie de elementos importantes sobre su correspondiente en la primera versión del producto, `ObjectQuery<TEntity>`. Estos elementos se exponen en los siguientes puntos:

- `ObjectSet<TEntity>` se basa en un contrato, especificado por la interfaz `IObjectSet<TEntity>` gracias a lo cual podemos facilitar enormemente las capacidades de testeo de nuestras capas de persistencia. Al basarse en un contrato podemos crear objetos simulados y así disponer de test realmente asilados.

- Incorporar a `ObjectQuery<TEntity>` capacidades para agregar y eliminar elementos por medio de los métodos `AddObject`, `DeleteObject`, gracias a lo cual la implementación de un patrón *Repository* es más natural.

Las definiciones completas de *ObjectSet<TEntity>* y *ObjectQuery<TEntity>* pueden verse a continuación en los siguientes fragmentos:

```
public class ObjectSet<TEntity> : ObjectQuery<TEntity>,
IObjectSet<TEntity>, IQueryable<TEntity>, IEnumerable<TEntity>,
IQueryable, IEnumerable where TEntity : class
{
        public EntitySet EntitySet { get; }

        public void AddObject(TEntity entity);

        public TEntity ApplyCurrentValues(TEntity currentEntity);

        public TEntity ApplyOriginalValues(TEntity originalEntity);

        public void Attach(TEntity entity);

        public T CreateObject<T>() where T : class, TEntity;

        public TEntity CreateObject();

        public void DeleteObject(TEntity entity);

        public void Detach(TEntity entity);
}


public class ObjectQuery<T> : ObjectQuery, IOrderedQueryable<T>,
IQueryable<T>, IEnumerable<T>, IOrderedQueryable, IQueryable,
IEnumerable, IListSource
{
        public ObjectQuery(string commandText, ObjectContext
context);
```

```
        public ObjectQuery(string commandText, ObjectContext context,
MergeOption mergeOption);


        public string Name { get; set; }

        public ObjectQuery<T> Distinct();

        public ObjectQuery<T> Except(ObjectQuery<T> query);

        public ObjectResult<T> Execute(MergeOption mergeOption);

        public ObjectQuery<DbDataRecord> GroupBy(string keys, string
projection, params ObjectParameter[] parameters);

        public ObjectQuery<T> Include(string path);

        public ObjectQuery<T> Intersect(ObjectQuery<T> query);

        public ObjectQuery<TResultType> OfType<TResultType>();

        public ObjectQuery<T> OrderBy(string keys, params
ObjectParameter[] parameters);

        public ObjectQuery<DbDataRecord> Select(string projection,
params ObjectParameter[] parameters);

        public ObjectQuery<TResultType>
SelectValue<TResultType>(string projection, params ObjectParameter[]
parameters);

        public ObjectQuery<T> Skip(string keys, string count, params
ObjectParameter[] parameters);

        public ObjectQuery<T> Top(string count, params
ObjectParameter[] parameters);
        public ObjectQuery<T> Union(ObjectQuery<T> query);

        public ObjectQuery<T> UnionAll(ObjectQuery<T> query);

        public ObjectQuery<T> Where(string predicate, params
ObjectParameter[] parameters);

}
```

La creación de objetos `ObjectSet<TEntity>` se realiza mediante la llamada al método `CreateObjectSet<TEntity>` que nos ofrecen nuestros contextos de trabajo. Este método, nos ofrece dos sobrecargas distintas. La primera de ellas sin parámetros de entrada, y la segunda con un parámetro solicitando en nombre del `EntitySet` asociado.

```
public ObjectSet<TEntity> CreateObjectSet<TEntity>(string
entitySetName)             where TEntity : class;

public void CreateProxyTypes(IEnumerable<Type> types);
```

Piense que este tipo es la puerta de entrada a los distintos `EntitySet` definidos en nuestros modelos. Generalmente (puede ver esto en los diferentes contextos de trabajo generados hasta ahora) se suele utilizar la primera sobrecarga, en la que el nombre del `EntitySet` asociado se infiere del tipo `TEntity` con el que se esté trabajando.

Una vez que se dispone de un objeto `ObjectSet<TEntity>`, se pueden aplicar consultas adicionales sobre el conjunto de resultados que ese objeto producirá utilizando **LINQ to Entities**, que no es otra cosa que una implementación de un proveedor de LINQ sobre EF. Observe en el fragmento anterior con la implementación de `ObjectQuery<TEntity>` como este tipo implementa las interfaces `IEnumerable<T>` e `IQueryable<T>`, lo que la hace plenamente capaz de servir como origen para las expresiones de consulta.

Nota: Este libro ofrece un apéndice completo sobre los fundamentos de LINQ, incluyendo todo lo necesario para comprender su uso y funcionamiento.

Una vez que se haya formulado una consulta de cualquiera de las dos formas antes comentadas, cuando el objeto `ObjectSet<TEntity>` se ejecute o se enumere lo primero que ocurrirá será que se enviará a la fuente de datos (traducida al dialecto específico del almacén) la sentencia correspondiente; los resultados de la consulta se obtendrán ("materializarán") en forma de objetos de .NET Framework. A continuación se presentan varios ejemplos basados en el uso de sentencias eSQL. Las consultas de *LINQ to Entities* las estudiaremos algo más adelante.

2.3.1.- Ejemplos de creación de consultas

Como hemos comentado antes, para la creación de objetos `ObjectSet<TEntity>` disponemos de un método en el contexto de trabajo. Sin embargo para trabajar con `ObjectQuery<TEntity>`, su clase base, disponemos de varias opciones. La primera de todas consiste en hacer uso de sus constructores, a los cuales debemos especificar el texto de la consulta que deseamos realizar y el contexto de trabajo que se va a utilizar. Recuerde que es éste último quien conoce todos los metadatos del modelo. La segunda de las opciones es usar el método `CreateQuery<T>` que también nos proporcionan los contextos de trabajo.

Para empezar a mostrar ejemplos de consultas partiremos de la siguiente estructura relacional, **sql\ef4_ex_8.sql**, en la que disponemos de las tablas *Product*, *Software* y *Book*.

```
--SQL\EF4_EX_8.sql
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Audio](
    [idAudio] [int] IDENTITY(1,1) NOT NULL,
```

```
    [title] [nvarchar](200) NOT NULL,
 CONSTRAINT [PK_Audio] PRIMARY KEY CLUSTERED
(
    [idAudio] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Song](
    [idSong] [int] NOT NULL,
    [Artist] [nvarchar](200) NOT NULL,
 CONSTRAINT [PK_Song_1] PRIMARY KEY CLUSTERED
(
    [idSong] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Podcast](
    [idPodCast] [int] NOT NULL,
    [podCastUrl] [nvarchar](50) NOT NULL,
    [podCastHits] [int] NOT NULL,
 CONSTRAINT [PK_Podcast] PRIMARY KEY CLUSTERED
(
    [idPodCast] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

ALTER TABLE [dbo].[Song]  WITH CHECK ADD  CONSTRAINT [FK_Song_Audio]
FOREIGN KEY([idSong])
REFERENCES [dbo].[Audio] ([idAudio])
GO
ALTER TABLE [dbo].[Song] CHECK CONSTRAINT [FK_Song_Audio]
GO

ALTER TABLE [dbo].[Podcast]  WITH CHECK ADD  CONSTRAINT
[FK_Podcast_Audio] FOREIGN KEY([idPodCast])
REFERENCES [dbo].[Audio] ([idAudio])
GO
ALTER TABLE [dbo].[Podcast] CHECK CONSTRAINT [FK_Podcast_Audio]
GO
```

Si recuerda, este mismo esquema relacional fue el propuesto en el ejemplo de herencia por tipo del capítulo 2. Haremos por lo tanto aquí un modelo *EDM* con esta jerarquía, y empezaremos a realizar nuestras primeras consultas, **code\EF4_EX_20**.

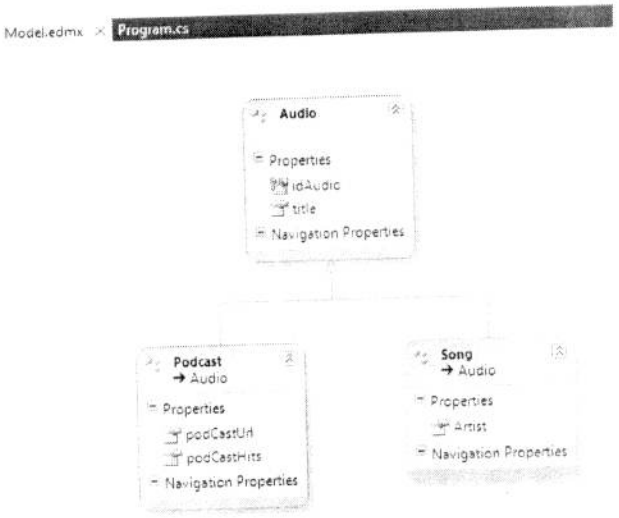


Figura 4.8.- Modelo de EDM del ejemplo

Tal y como ya hemos comentado, el asistente, además de la creación de un contexto de trabajo (el contenedor de entidades), también crea una clase para cada una de las entidades del mismo. De hecho, hemos visto diferentes opciones para su generación, clases prescriptivas, *STE* u objetos *POCO*. Veamos cómo usar el contenedor de entidades para obtener las entidades de tipo `Audio` por medio de un objeto `ObjectQuery<Audio>`:

```
using (EF4EXEntities context = new EF4EXEntities())
{
    string eSQL = "SELECT VALUE p FROM
OFTYPE(EF4EXEntities.Audios,EF4_EX_20.Audio) AS p";

    ObjectQuery<Audio> query = new ObjectQuery<Audio>(eSQL,
context);

    //realizar consulta aquí
}
```

Observe la diferencia de esta consulta con respecto a la realizada en el capítulo anterior cuando hablamos de consultas polimórficas en *Entity Client*. La expresión `OFTYPE` en este caso se refiere a un tipo concreto de .NET, por lo que está cualificada completamente, en lugar de utilizarse simplemente al *namespace* del modelo *EDM*.

Nota: ObjectQuery<TEntity> nos permite permite definir la consulta estableciendo directamente el valor de la cláusula FROM. Por ello, para el ejemplo del listado anterior hubiera bastado simplemente con:

```
query = new
ObjectQuery<Audio>("OFTYPE(EF4EXEntities.Audios,EF4_EX_20.Audio)",
context);
```

Una vez creada la consulta con la que deseamos trabajar fíjese en lo sencillo de su ejecución en comparación con la vía que nos propone Entity Client, ya que podemos obtener los resultados de la misma directamente de forma tipada por medio de nuestras clases de entidades. Para obtener los resultados, podemos simplemente enumerar la consulta y navegar por sus resultados:

```
using (EF4EXEntities context = new EF4EXEntities())
{

      ObjectQuery<Audio> query = new
ObjectQuery<Audio>("OFTYPE(EF4EXEntities.Audios,EF4_EX_20.Audio)",
context);

      //ejecuta la consulta
      foreach (Audio item in query)
          //TODO...

}
```

En realidad, en el momento en el que un objeto `ObjectQuery` es enumerado se produce una llamada a su método `Execute`, el encargado de lanzar la consulta específica a la base de datos subyacente; éste devuelve los resultados en un objeto enumerable de tipo `ObjectResult<TEntity>`.

```
using (EF4EXEntities context = new EF4EXEntities())
{
                ObjectQuery<Audio> query = new
ObjectQuery<Audio>("OFTYPE(EF4EXEntities.Audios,EF4_EX_20.Audio)",
context);

                ObjectResult<Audio> result =
query.Execute(MergeOption.AppendOnly);

                foreach (Audio item in result)
                {
                    //TODO...
                }

}
```

Por supuesto, `ObjectQuery` nos ofrece las mismas posibilidades de las que disponíamos con Entity Client, como por ejemplo la utilización de consultas parametrizadas. En este caso, los parámetros ya no se especificarán mediante el tipo

`EntityParameter`, sino por medio de un nuevo tipo denominado `ObjectParameter` (definido en `System.Data.Objects`), cuyo uso es idéntico al anterior, tal y como podemos constatar a continuación.

```
using (EF4EXEntities context = new EF4EXEntities())
{
                string eSQL = "SELECT VALUE p FROM
OFTYPE(EF4EXEntities.Audios,EF4_EX_20.Audio)";
                eSQL +=" AS p ORDER BY p.idAudio SKIP @skip LIMIT
@limit";
                ObjectQuery<Audio> query = new
ObjectQuery<Audio>(eSQL, context);
                query.Parameters.Add(new ObjectParameter("skip", 0));
                query.Parameters.Add(new ObjectParameter("limit",
10));
                ObjectResult<Audio> result =
query.Execute(MergeOption.AppendOnly);
                foreach (Audio item in result)
                {
                    //TODO...
                }
}
```

Si desea revisar las consultas generadas y lanzadas contra la base de datos, `ObjectQuery` dispone, de igual manera que `EntityConnection`, de un método llamado `ToTraceString` que nos devolverá una cadena de caracteres con el texto de la consulta en el dialecto específico del almacén de datos con el que estemos trabajando.

2.3.2.- Métodos de construcción de consultas

Seguro que estará de acuerdo con nosotros si decimos que la mayoría de las consultas que se presentan en la vida real no son tan simples como consultar una entidad, aun cuando ésta pertenezca a una determinada jerarquía, como ha ocurrido en los ejemplos anteriores. La mayoría de las consultas incluyen sentencias de filtro, ordenación, agrupación, etc. Desde el equipo de trabajo de EF, con el fin de facilitarnos el trabajo y aumentar nuestra productividad, han puesto a nuestra disposición toda una serie de métodos de construcción de consultas, conocidos en su conjunto como **SQL Builder Methods**. Estos métodos de construcción operan sobre las expresiones de consulta más comunes y cada uno de ellos tiene asociada su correspondiente expresión de eSQL. La siguiente tabla nos muestra los distintos métodos de construcción disponibles y su equivalente en el lenguaje de consultas para modelos conceptuales.

Tabla 1.- Métodos de construcción de consultas

Método	Expresión eSQL
Distinct	DISTINCT
Except	EXCEPT

GroupBy	GROUP BY
Intersect	INTERSECT
OfType	OFTYPE
OrderBy	ORDER BY
Select	SELECT
SelectValue	SELECT VALUE
Skip	SKIP
Top	TOP y LIMIT
Union	UNION
UnionAll	UNION ALL
Where	WHERE

Antes de comenzar a ver ejemplos sobre los métodos de construcción de consultas es necesario indicar que, al igual que nosotros establecemos un alias dentro de la expresión FROM (en el ejemplo anterior este alias es `p`), por defecto los objetos `ObjectQuery<TEntity>` para una determinada consulta en la que únicamente se especifique la cláusula FROM siempre establecen el alias `it`, nombre que deberemos utilizar en tales casos para acceder a los elementos de la consulta, como iremos viendo a lo largo de esta sección.

Utilizaremos como origen para las consultas la única propiedad de tipo `ObjectSet<TEntity>`, subclase de `ObjectQuery<TEntity>` generada automáticamente dentro de la clase del contexto para el modelo de la figura 4.8. Si recuerda lo dicho unos párrafos anteriores, el contexto de trabajo generado a partir de un modelo crea, para cada uno de los `EntitySet`, una propiedad pública que nos devuelve un `ObjectSet<TEntity>` específico del `EntitySet`.

```
ObjectSet<Audio> query = dbContext.Audios;
```

❖ OfType

En nuestro primer ejemplo de objetos de consulta, vimos cómo construir una consulta que nos devolvía el conjunto de entidades `Audio` contenidas en nuestro almacén de datos. Para esta entidad, como acabamos de comentar, al ser raíz de una jerarquía, el contexto de trabajo dispone de una propiedad de consulta. Sin embargo, es más que probable que necesitamos realizar alguna consulta sobre alguno de los elementos de su jerarquía como `Song`. Puesto que para las entidades de una jerarquía no tenemos propiedades como la anterior ¿cómo podemos entonces a partir de `Audio` acceder a las entidades de tipo `Song`? La respuesta es realmente sencilla: entre los distintos métodos de construcción de consultas disponemos de uno, `OfType`, que corresponde a la expresión OFTYPE, la misma que hemos utilizado en la consulta eSQL. Por lo tanto, la manera de obtener las entidades de tipo `Song` sería la siguiente, **code\ef4_ex_21**.

```
using (EF4EXEntities dbContext = new EF4EXEntities())
{
```

```
        ObjectQuery<Song> query = dbContext.Audios.OfType<Song>();
    }
```

Nota: Le recomendamos que tenga especial cuidado con el uso de OfType. En modelos conceptuales complejos, y más si los tipos de la jerarquía, como es lógico, dispone de asociaciones también complejas, el uso de la herencia podría complicar fuertemente las consultas enviadas al motor de base de datos.

❖ Where

El método de construcción `Where` es el más trivial de todos, por su similitud con la expresión **WHERE** en la sentencia **SELECT** de los distintos dialectos de SQL. Supongamos que usted desea restringir la lista de las entidades `Song` a obtener a aquellas que contengan como primer carácter en la propiedad `Titulo` la letra T. La implementación de esta consulta es tan sencilla como una llamada al método `Where`, indicando la condición a aplicar en forma de una cadena de caracteres, como sigue:

```
using (EF4EXEntities dbContext = new EF4EXEntities())
{
    var query = dbContext.Audios
                         .Where("LEFT(it.title,1)='T'");

}
```

Fíjese en cómo las llamadas a métodos de construcción de consultas pueden encadenarse, facilitándose así la lectura y comprensión de la consulta.

En el caso de que el método de construcción sea susceptible de ser parametrizado con el valor de alguna variable, podemos incluir en las llamadas a los mismos la lista de parámetros a utilizar. Podríamos parametrizar la consulta anterior para obtener los elementos de tipo `Audio` cuyo primer carácter coincida con el valor de un determinado parámetro, de la siguiente forma:

```
using (EF4EXEntities dbContext = new EF4EXEntities())
{
    Var query = dbContext.Audios
                         .Where("LEFT(it.title,1)=@letter",new
ObjectParameter("letter","T"));

}
```

En líneas anteriores comentamos que el alias por defecto para los objetos de consulta era `it`. Como cabría esperar, este alias puede ser modificado a nuestro gusto. Para ello basta con establecer en la propiedad `Name` de nuestro objeto de consulta el alias que deseemos. Así, podríamos haber expresado la consulta anterior de la siguiente forma:

```
using (EF4EXEntities dbContext = new EF4EXEntities())
{
     ObjectQuery<Audio> query = dbContext.Audios;
     query.Name = "audio";
     query = query.Where("LEFT(audio.title,1)=@letter", new
ObjectParameter("letter", "T"));

}
```

❖ Intersect

Este método de construcción de consultas permite limitar los resultados de la ejecución de una consulta a los encontrados dentro de otra consulta o de un conjunto enumerable del mismo tipo. Si quisiéramos limitar los resultados de la consulta anterior a solo aquellas entidades `Audio` para los que el valor de la propiedad `idAudio` fuera mayor que 1000, podríamos utilizar este método de construcción de la siguiente forma:

```
using (EF4EXEntities dbContext = new EF4EXEntities())
{
      ObjectQuery<Audio> queryA = dbContext.Audios;
      queryA = queryA.Where("LEFT(it.title,1)=@letter", new
ObjectParameter("letter", "T"));

       var queryB = dbContext.Audios
                             .Where("it.idAudio > 1000");

        var result = queryA.Intersect(queryB);
}
```

❖ Except

Al igual que el método anterior, `Except` también permite limitar los resultados de una consulta determinada, en este caso excluyendo los elementos pertenecientes a otra consulta o conjunto enumerable. En el siguiente ejemplo, el resultado de la consulta serían todas las entidades de tipo `Audio` cuyo primer carácter en la propiedad `Titulo` fuera T, exceptuando aquellas entidades que tuvieran un valor superior a 1050 en su propiedad `idAudio`.

```
using (EF4EXEntities dbContext = new EF4EXEntities())
{
       ObjectQuery<Audio> queryA = dbContext.Audios;
       queryA = queryA.Where("LEFT(it.title,1)=@letter", new
ObjectParameter("letter", "T"));

       var queryB = dbContext.Audios
                             .Where("it.idAudio > 1050");

       var result = queryA.Except(queryB);
}
```

❖ Union

`Union` permite combinar el resultado de dos consultas diferentes, sin repetir los elementos duplicados. Por ejemplo, si quisiéramos obtener las entidades `Audio` cuyo código fuera mayor que 1050 o su propiedad `Titulo` comenzara por A, podríamos hacer algo similar a lo siguiente:

```
using (EF4EXEntities dbContext = new EF4EXEntities())
{
    ObjectQuery<Audio>queryA = dbContext.Audios;
    queryA = queryA.Where("LEFT(it.title,1)=@letter", new
ObjectParameter("letter", "A"));

    var queryB = dbContext.Audios
                          .Where("it.idAudio > 1050");

    var result = queryA.Union(queryB);
}
```

❖ UnionAll

Al contrario que `Union`, el método `UnionAll` realiza una unión de los resultados de dos consultas, pero en este caso admitiendo resultados duplicados. En algunas ocasiones es necesario recurrir obligatoriamente a este método en vez del anterior, puesto que algunos tipos de datos pueden no ser comparables, como por ejemplo en el caso del tipo de datos `Image`. En el caso de que una consulta con una llamada al método `Union` se ejecute y la entidad a consultar contenga algún tipo de datos no comparable, ésta fallará en tiempo de ejecución y deberemos proceder a sustituirla por una llamada a `UnionAll`, de la siguiente forma:

```
using (EF4EXEntities dbContext = new EF4EXEntities())
{
    ObjectQuery<Audio> queryA = dbContext.Audios;
    queryA = queryA.Where("LEFT(it.title,1)=@letter", new
ObjectParameter("letter", "A"));

    var queryB = dbContext.Audios
                          .Where("it.idAudio > 1050");

    var result = queryA.UnionAll(queryB);
}
```

❖ Distinct

Este método, cuyo equivalente en eSQL es `SELECT DISTINCT`, permite obtener una combinación de dos consultas en la que todos los elementos obtenidos son diferentes.

Al igual que en el caso de `Union`, debemos asegurarnos que todos los miembros de la entidad a consultar sean de tipos comparables. En realidad, el método de construcción `Union` utiliza internamente `Distinct` para eliminar los elementos duplicados.

```
using (EF4EXEntities dbContext = new EF4EXEntities())
{
     ObjectQuery<Audio> queryA = dbContext.Audios;
     queryA = queryA.Where("LEFT(it.title,1)=@letter", new
ObjectParameter("letter", "A"));

     var queryB = dbContext.Audios
                          .Where("it.idAudio > 1050");

     var result = queryA.Union(queryB).Distinct();
}
```

❖ OrderBy

Como seguramente usted ya imagina, el método de construcción `OrderBy` nos permite especificar el orden en el que vamos a obtener los resultados de una determinada consulta. Podemos ver su utilización en el ejemplo siguiente:

```
using (EF4EXEntities dbContext = new EF4EXEntities())
{
    ObjectQuery<Audio> queryA = dbContext.Audios;
    queryA = queryA.Where("LEFT(it.title,1)=@letter", new
ObjectParameter("letter", "A"));

    var queryB = dbContext.Audios
                         .Where("it.idAudio > 1050")
                         .OrderBy("it.idAudio");
}
```

❖ GroupBy

El método `GroupBy` permite agrupar los resultados de una consulta en función de un criterio determinado. La sintaxis de este método es ligeramente más compleja que la de los métodos anteriores. En primer lugar es necesario especificar cuál es el criterio de agrupación; es decir, especificar en función de qué propiedad o propiedades deseamos crear los distintos grupos, y posteriormente especificar qué elementos vamos a seleccionar para cada uno de nuestros grupos.

En el siguiente fragmento de código se presenta un ejemplo de consulta que hace uso de este método de construcción para obtener la cantidad de elementos `Song` que hay para cada uno de los valores de la propiedad `Artist`.

```
using (EF4EXEntities dbContext = new EF4EXEntities())
{
     string key ="it.Artist";
     string proj = "Count(it.idAudio) AS number";
```

```
        var queryA = dbContext.Audios
                              .OfType<Song>()
                              .GroupBy(key,proj);
    }
```

Fíjese en el ejemplo anterior como el tipo específico de nuestra consulta en este caso no se corresponde con ninguna de las entidades de nuestro modelo. El resultado de la consulta anterior no produce entidades de tipo `Song`, sino solamente un resultado con la cantidad de elementos de tipo `Song`. En este caso la materialización de los resultados se hará sobre objetos `DbDataRecord`, tipo que implementa la interfaz `IExtendedDataRecord`.

❖ Skip

El método `Skip` permite ordenar una consulta en función de un determinado criterio y posteriormente saltar un número de resultados determinado. Si quisiéramos obtener las entidades `Audio` de nuestro almacén de base de datos ordenadas por su propiedad `Titulo`, y no tomar en consideración los 30 primeros resultados, podríamos hacer lo siguiente:

```
using (EF4EXEntities dbContext = new EF4EXEntities())
{
    ObjectQuery<Audio> queryA = dbContext.Audios
                                         .Skip("it.titulo", "@skip",
new ObjectParameter("skip", 30));
}
```

❖ Top

`Top` nos permite limitar la cantidad de resultados de una consulta a un número determinado de elementos. El siguiente ejemplo muestra cómo obtener los 10 primeros elementos de una consulta sobre las entidades `Audio`.

```
using (EF4EXEntities dbContext = new EF4EXEntities())
{
    ObjectQuery<Audio> queryA = dbContext.Audios
                                         .Top("10");

}
```

Como hemos mostrado en la tabla de métodos, `Top` puede producir distintas expresiones en eSQL. En nuestro ejemplo anterior, la aparición del método `Top` provocará la generación de la cláusula `TOP` en la sentencia eSQL. Sin embargo, si combinamos los métodos `Skip` y `Top`, se generará la cláusula `LIMIT`. Esto nos facilita enormemente la paginación de los resultados de las consultas.

```
using (EF4EXEntities dbContext = new EF4EXEntities())
{
    ObjectQuery<Audio> queryA = dbContext.Audios
                                         .Skip("it.titulo", "@skip",
new ObjectParameter("skip", 30))
                                         .Top("@top",new
ObjectParameter("top",10));

}
```

2.3.3.- Navegación entre entidades

Hasta ahora hemos visto algunos ejemplos de consultas sobre nuestro tipo `ObjectSet<TEntity>`, muchos ejemplos apoyados en el elemento que le da soporte `ObjectQuery<TEntity>`. Sin embargo, ninguna de esas consultas implicaba una navegación entre las entidades presentes en un modelo. A lo largo de este punto y del siguiente, trataremos de mostrar las posibilidades de navegación entre entidades y el concepto de expansión de consultas. Para esta tarea nos apoyaremos en el modelo relacional siguiente, el cual consta de tres tablas simples, `Customer`, `Order` y `OrderDetail`.

```
--sql\EF4_EX_15.sql

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Customer](
    [idCustomer] [int] NOT NULL,
    [firstName] [nvarchar](200) NOT NULL,
    [lastName] [nvarchar](200) NOT NULL,
    [age] [int] NOT NULL,
 CONSTRAINT [PK_Customer] PRIMARY KEY CLUSTERED
(
    [idCustomer] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Order](
    [idOrder] [int] NOT NULL,
    [idCustomer] [int] NOT NULL,
    [orderDate] [datetime] NOT NULL,
    [totalOrder] [decimal](18, 0) NOT NULL,
 CONSTRAINT [PK_Order] PRIMARY KEY CLUSTERED
(
    [idOrder] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
```

```
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[OrderDetail](
    [idOrderDetail] [int] NOT NULL,
    [idOrder] [int] NOT NULL,
    [quantity] [int] NOT NULL,
    [price] [decimal](18, 0) NULL,
 CONSTRAINT [PK_OrderDetail] PRIMARY KEY CLUSTERED
(
    [idOrderDetail] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

ALTER TABLE [dbo].[Order]  WITH CHECK ADD  CONSTRAINT
[FK_Order_Customer] FOREIGN KEY([idCustomer])
REFERENCES [dbo].[Customer] ([idCustomer])
GO
ALTER TABLE [dbo].[Order] CHECK CONSTRAINT [FK_Order_Customer]
GO

ALTER TABLE [dbo].[OrderDetail]  WITH CHECK ADD  CONSTRAINT
[FK_OrderDetail_Order] FOREIGN KEY([idOrder])
REFERENCES [dbo].[Order] ([idOrder])
GO
ALTER TABLE [dbo].[OrderDetail] CHECK CONSTRAINT
[FK_OrderDetail_Order]
GO
```

Una vez que tenemos el fragmento SQL anterior crearemos, a partir de él, un modelo de entidades para realizar nuestros ejemplos, **code\ef4_ex_22,** por simplicidad no tocaremos el modelo creado, ni nombres de entidades ni de propiedades.

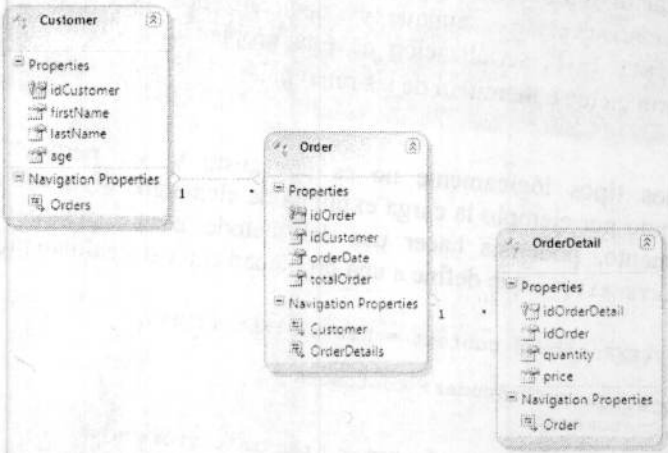


Figura 4.9.- Modelo de entidades para nuestros ejemplos

Como comentamos en el capítulo dos, las asociaciones de entidades llevan aparejadas *propiedades de navegación*. Estas propiedades de navegación forman parte, lógicamente, del código generado para nuestras entidades y por lo tanto su representación en código dependerá del artefacto de generación seleccionado.

En el artefacto de generación por defecto, clases prescriptivas, las propiedades de navegación, sean de tipo colección o no, se basan en nuevos tipos definidos por la infraestructura de Entity Framework. Así, por ejemplo, la propiedad de navegación que relaciona nuestras entidades `Customer` y `Order` se define como se muestra en el siguiente fragmento.

```
/// <summary>
/// No Metadata Documentation available.
/// </summary>
[XmlIgnoreAttribute()]
[SoapIgnoreAttribute()]
[DataMemberAttribute()]
[EdmRelationshipNavigationPropertyAttribute("EF4_EXModel",
"FK_Order_Customer", "Order")]
public EntityCollection<Order> Orders
{
            get
            {
                return
((IEntityWithRelationships)this).RelationshipManager.GetRelatedCollection
<Order>("EF4_EXModel.FK_Order_Customer", "Order");
            }
            set
            {
                if ((value != null))
                {

((IEntityWithRelationships)this).RelationshipManager.InitializeRelatedCol
lection<Order>("EF4_EXModel.FK_Order_Customer", "Order", value);
                }
            }
}
```

Nota: Los atributos `XmlIgnore` y `SoapIgnore` permiten "sacar" las propiedades de navegación de una serialización con XmlSerializer puesto que podríamos entrar en serializaciones de ciclos, algo que este elemento no soporta. Por el contrario, `DataMemberAttribute`, aunque es opcional, nos permite indicar a `DataContractSerializar` la serialización de esta propiedad, recuerde que este serializador soporta ciclos e identidad de las referencias.

El uso de estos tipos lógicamente no es por gusto y nos permiten algunas funcionalidades como por ejemplo la carga explícita de elementos. Así, como se ve en el siguiente fragmento, podemos hacer uso del método `Load` incluído en el tipo `EntityCollection<TEntity>`, que define a una propiedad con cardinalidad plural.

```
using (EF4EXEntities context = new EF4EXEntities())
{
      ObjectQuery<Customer> customers =
context.Customers;
                foreach (Customer item in customers)
                    //carga explícita
                    item.Orders.Load();

}
```

Tanto `EntityCollection<TEntity>` como `EntityReference<TEntity>`, tipo para las propiedades de cardinalidad simple, comparten una interfaz de trabajo común, denominada `IRelatedEnd`, que forma la base del sistema de relaciones entre entidades a nivel de objetos. Esta interfaz, cuya estructura se puede observar en el siguiente listado, contiene los elementos básicos para que nuestras entidades puedan acceder a las relaciones que posean usando para ello el método `Load`, el cual se encarga de solicitar la entidad o entidades relacionadas como hemos viso justamente en el ejemplo anterior.

```
public interface IRelatedEnd
{
        bool IsLoaded { get; }

        string RelationshipName { get; }
        RelationshipSet RelationshipSet { get; }

        string SourceRoleName { get; }

        string TargetRoleName { get; }

        void Add(object entity);

        IEnumerable CreateSourceQuery();

        void Load();
```

```
        void Load(MergeOption mergeOption);

        bool Remove(IEntityWithRelationships entity);

        bool Remove(object entity);

        // Simplificada…
}
```

Nota: Para el caso de las asociaciones de cardinalidad simple, propiedades de navegación hacían un único elemento, además de la propiedad `EntityReference<TEntity>` correspondiente, ADO.NET Entity Framework genera una propiedad referenciando al tipo en cuestión, `TEntity`, al que solamente podremos acceder si está previamente cargado.

El elemento gracias al cual podemos acceder a los distintos `IRelatedEnd` de una entidad está representado por el tipo `RelationshipManager`, accesible a todas las entidades, al heredar éstas de la clase base `EntityObject`, que como ya sabemos, a su vez implementa la interfaz `IEntityWithRelationships`. En el siguiente ejemplo de código se muestra cómo consultar esta clase `RelationshipManager` para obtener las relaciones de una entidad `Customer` y cómo realizar la navegación a través de las mismas.

```
static void RelationshipManager()
{
   using (EF4EXEntities context = new EF4EXEntities())
   {
       ObjectQuery<Customer> customers = context.Customers;

       foreach (Customer item in customers)
          ViewAndLoadRelation(item);

   }
}

static void ViewAndLoadRelation(Customer item)
{
            RelationshipManager relationManager =
((IEntityWithRelationships)item).RelationshipManager;

            IEnumerable<IRelatedEnd> relatedEnds =
relationManager.GetAllRelatedEnds();
            foreach (IRelatedEnd related in relatedEnds)
            {
                Console.WriteLine("Relation Name:{0} SourceRole:{1}
TargetRole:{2}",
                                  related.RelationshipName,
                                  related.SourceRoleName,
                                  related.TargetRoleName);

                related.Load();
            }
}
```

Puede comparar la similitud del trabajo realizado aquí al propuesto por las propiedades de navegación creadas por el generador de entidades. Este ejemplo no es más que un simple ejercicio para que comprenda el código generado; algo que no tendremos nunca que realizar, puesto que esta tarea ya la tenemos implementada y nos bastará con hacer uso de las propiedades, tal y como vimos anteriormente.

Llegados a esta parte de la sección que nos ocupa, ya sabemos cómo a partir de las propiedades de navegación podemos cargar los elementos asociados mediante una relación con una determinada entidad: a través del método `Load`. Este método `Load` es en realidad **una implementación explícita del patrón Lazy Loading,** puesto que ninguna relación se carga hasta que es realmente necesario. Con el fin de asegurarnos que podemos acceder al elemento o elementos de una determinada relación, la interfaz `IRelatedEnd` pone a nuestra disposición una propiedad, `IsLoaded`, que nos devuelve un valor verdadero o falso para indicarnos si podemos acceder a los elementos de la relación. En el siguiente ejemplo, **code\ef4_ex_23**, podemos ver un ejemplo de uso.

```
private static void IsLoaded()
{
    using (EF4EXEntities context = new EF4EXEntities())
    {
        ObjectQuery<Customer> customers = context.Customers;

        foreach (Customer item in customers)
        {
            if (item.Orders.IsLoaded)
            {
                foreach (Order order in item.Orders)
                    Console.WriteLine("Order Date :{0}",
order.orderDate);
                }
            }
            Console.ReadLine();
        }
}
```

Además de esta implementación explícita del patrón Lazy Load en `IRelatedEnd` con el método `Load`. ADO.NET Entity Framework 4.1 dispone en esta nueva versión la posibilidad de uso de la carga perezosa de forma implícita o **Lazy Load tradicional**.

Para activar esta posibilidad dentro de nuestro contexto de trabajo tendremos que activar su uso dentro de las opciones del contexto, opciones representadas por una propiedad llamada `ContextOptions`, la cual devuelve una instancia del tipo `ObjectContextOptions`.

```
public sealed class ObjectContextOptions
{
public bool LazyLoadingEnabled { get; set; }

public bool ProxyCreationEnabled { get; set; }

public bool UseLegacyPreserveChangesBehavior { get; set; }
}
```

Como puede imaginar, la propiedad `LazyLoadingEnabled` nos permitirá establecer si deseamos o no habilitar esta característica de carga perezosa de forma implícita. En el siguiente ejemplo, también disponible en **code\ef4_ex_23**, vemos un ejemplo de uso en el que se accede a la propiedad `Orders` sin necesidad de realizar una carga previa con el método `Load`.

```
using (EF4EXEntities context = new EF4EXEntities())
{
      //set lazy loading enabled
      context.ContextOptions.LazyLoadingEnabled = true;

      ObjectQuery<Customer> customers = context.Customers;

      foreach (Customer item in customers)
      {
            if (item.Orders.IsLoaded)
            {
                foreach (Order order in item.Orders)
                      Console.WriteLine("Order Date :{0}",
order.orderDate);
            }
       }
       Console.ReadLine();
}
```

Hasta ahora hemos visto las características de nuestras propiedades en las clases prescriptivas, clases por defecto en ADO.NET Entity Framework. Lógicamente como sabemos además de la generación por defecto también disponemos de nuestras entidades **STE** y **POCO**. Para el caso de nuestras entidades **STE, code\ef4_ex_24,** mientras que las relaciones de cardinalidad simple, hacia una entidad no una colección, no son más que propiedades normales hacia el tipo de la asociación. En el caso de las asociaciones *a muchos*, se definen mediante el tipo `TrackableCollection<TEntity>`.

Este tipo se define conjuntamente con nuestras entidades en la plantilla creada una vez que hemos seleccionado el artefacto. La tarea principal de este tipo, cuya definición vemos a continuación, es la de ofrecer una colección capaz de notificar cambios en la misma, además de manejar la posible duplicación de elementos.

```
public class TrackableCollection<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);
        }
     }
}
```

Puesto que este tipo no tiene que ver nada con EF (no es ningún tipo dependiente de ADO.NET Entity Framework), la solución de carga perezosa explícita como en nuestras clases prescriptivas no está disponible. De hecho, aunque le pueda parecer extraño tampoco la opción de **Lazy Loading implícita** es recomendable para este artefacto de generación. La razón es en realidad muy simple, estas entidades están pensadas para trabajar en modelos distribuídos. Ahondaremos en sus capacidades para replicar en el contexto los cambios realizados sobre las mismas de forma desconectada más adelante. Por lo tanto, estas entidades están pensadas para serializarse. Si tuviéramos activado **Lazy Loading** de forma implícita cuando la entidad se serializara se ejecutarían todas las propiedades de navegación, puesto que la serialización invoca a todos los métodos `get` de las propiedades para obtener el valor a poner en el **stream** correspondiente.

Nota: La plantilla de generación del contexto de trabajo en Selft Tracking Entities establece la opción LazyLoadingEnabled a false. Aunque la recomendación lógica es no usar Lazy Loading técnicamente no tendría ningún impedimento para realizarlo.

La última revisión de las propiedades de navegación la haremos de la plantilla de objetos **POCO**, **code\ef4_ex_25**, de la que hablamos en este mismo capítulo anteriormente. En esta plantilla nuestra propiedad `Orders` se definirá con el tipo genérico `ICollection<TEntity>`. Aunque internamente, como podemos ver en el código de esta propiedad, en realidad se trabaja con una colección de tipo *FixupCollection<TEntity>*.

```
public virtual ICollection<Order> Orders
{
    get
    {
        if (_orders == null)
        {
            var newCollection = new FixupCollection<Order>();
            newCollection.CollectionChanged += FixupOrders;
            _orders = newCollection;
        }
        return _orders;
    }

    set
    {
        if (!ReferenceEquals(_orders, value))
        {
            var previousValue = _orders as
FixupCollection<Order>;
            if (previousValue != null)
            {
                previousValue.CollectionChanged -= FixupOrders;
            }
            _orders = value;
```

```
                var newValue = value as FixupCollection<Order>;
                if (newValue != null)
                {
                    newValue.CollectionChanged += FixupOrders;
                }
            }
        }
    }
```

Si viéramos el código del tipo `FixupCollection<TEntity>` veríamos que, excepto el nombre, es idéntico al de `TrackableCollection<TEntity>`, entonces *¿Por qué nombres distintos?*. La razón principal a la pregunta anterior viene del propósito de esta colección, propósito que en nuestros objetos POCO será el de, además del almacén de los elementos asociados, mantener coherentes las asociaciones. A continuación vemos el método `FixupOrders`, método asociado al evento `CollectionChanged` de `FixupCollection`.

```
private void FixupOrders(object sender,
NotifyCollectionChangedEventArgs e)
{
    if (e.NewItems != null)
    {
        foreach (Order item in e.NewItems)
        {
            item.Customer = this;
        }
    }

    if (e.OldItems != null)
    {
        foreach (Order item in e.OldItems)
        {
            if (ReferenceEquals(item.Customer, this))
            {
                item.Customer = null;
            }
        }
    }
}
```

Como se puede deducir de este método, tras unas pequeñas observaciones, el propósito del mismo es el de garantizar que los elementos de tipo `Order` agregados a la colección tiene su propiedad de navegación inversa, `Customer`, establecida de forma correcta.

Por supuesto, la posibilidad de utilizar **Lazy Loading** de forma implícita también está disponible sin ningún tipo de restricción en este tipo de entidades.

2.3.4.- Expansión de consultas

En el punto anterior hemos visto como se crean las distintas propiedades de navegación para cada uno de los artefactos de las entidades, y cómo funciona la carga

perezosa, tanto de forma implícita como explícita. En ciertos escenarios, el uso de la carga perezosa no siempre es la mejor práctica. Imaginemos que estamos en el ejemplo anterior de entidades POCO y decidimos realizar el siguiente trabajo.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    context.ContextOptions.LazyLoadingEnabled = true;

    ObjectQuery<Order> orders = context.Orders;

    foreach (Order item in orders)
    {
        // realiza una consulta
        Console.WriteLine("FirstName:{0}", item.Customer.firstName);
    }
}
```

Si observamos con un profiler la ejecución del código anterior veríamos como el número de consultas enviadas a la base de datos sería de `N+1`, siendo N la cantidad de pedidos obtenidos en la primera consulta. Lógicamente esta no es una situación ideal, puesto que si conocemos a priori que vamos a acceder a nuestras entidades `Customer`, sería mucho más efectivo realizar una operación de mezcla en la base de datos que me permitiera obtener las entidades pedido con sus clientes asociados. Para realizar esta tarea `ObjectQuery<TEntity>` define un método de expansión de consultas llamado `Include`, a continuación en el siguiente fragmento:

```
public ObjectQuery<T> Include(string path)
```

`Include` nos permitirá definir la expansión que queramos realizar, su sintaxis es la siguiente:

```
Include(<ruta>)
```

donde `<ruta>` es la especificación de los nombres de las entidades (separadas por el carácter punto (.) en el caso de rutas complejas) participantes en relaciones y que se desean incluir en la consulta.

Si quisiéramos, por lo tanto, mejorar la consulta del anterior, podríamos expandir la consulta inicial sobre clientes con el fin de indicar que también vamos a usar los clientes de los pedidos reduciendo el número de consultas de N+1 a 1. A continuación puede ver tanto el código como la consulta generada.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    ObjectQuery<Order> orders = context.Orders.Include("Customer");

    foreach (Order item in orders)
    {
        // realiza una consulta
        Console.WriteLine("FirstName:{0}", item.Customer.firstName);
    }
}
```

```
SELECT
[Extent1].[idOrder] AS [idOrder],
[Extent1].[idCustomer] AS [idCustomer],
[Extent1].[orderDate] AS [orderDate],
[Extent1].[totalOrder] AS [totalOrder],
[Extent2].[idCustomer] AS [idCustomer1],
[Extent2].[firstName] AS [firstName],
[Extent2].[lastName] AS [lastName],
[Extent2].[age] AS [age]
FROM  [dbo].[Order] AS [Extent1]
INNER JOIN [dbo].[Customer] AS [Extent2] ON [Extent1].[idCustomer] =
[Extent2].[idCustomer]
```

Si usted ha revisado el nuevo código con el método `Include` probablemente le llamara la atención el hecho de tener que *hardcodear* un `string` con el nombre de la propiedad de navegación. Realmente esto podría ser un problema en modelos con una frecuencia de cambio buena, puesto que un cambio en el nombre de una propiedad de navegación podría implicar un error si el mismo se usara dentro de un método `Include`. Por suerte, esta *deficiencia* podríamos resolverla simplemente con la creación de un método extensor como el siguiente, definido a continuación:

```
static class IncludeExtensions
{
        public static ObjectQuery<TEntity> Include<TEntity>(this
ObjectQuery<TEntity> queryable, Expression<Func<TEntity, object>> path)
        where TEntity : class
        {
            return
queryable.Include(AnalyzeExpressionPath<TEntity,object>(path));
        }

        static string AnalyzeExpressionPath<TEntity,
S>(Expression<Func<TEntity, S>> expression)
        where TEntity : class
        {
            if (expression == (Expression<Func<TEntity, S>>)null)
                throw new ArgumentNullException("Invalid expression");

            MemberExpression body = expression.Body as MemberExpression;
            if (
                    (
                    (body == null)
                    ||

!body.Member.DeclaringType.IsAssignableFrom(typeof(TEntity))
                    )
                    ||
                    (body.Expression.NodeType !=
ExpressionType.Parameter))
            {
                throw new ArgumentException("Invalid path");
            }
            else
                return body.Member.Name;
        }
}
```

Una vez definido este nuevo método extensor, el uso del método `Include` podría establecerse como sigue:

```
ObjectQuery<Order> orders =
    context.Orders.Include(o=>o.Customer);
```

2.4.- LINQ To Entities

Para comprender esta sección es necesario que usted disponga de conocimientos sobre las novedades introducidas en la versión 3.0 de C#, y en particular las relacionadas con las consultas integradas en el lenguaje (LINQ). Para ello, el primer apéndice de este libro le ofrece un número considerable de páginas dedicadas a presentar estos fundamentos.

Nota: En esta misma editorial dispone de un completo libro sobre LINQ si necesita profundizar en el tema. Localízelo en nuestra tienda on-line: http://shop.campusmvp.com.

Como habrá podido observar, el salto que va de la forma de trabajar con Entity Client (Capítulo 3) a la basada en los servicios de objetos que estamos viendo a lo largo de este capítulo es muy grande. Desde la materialización de los resultados hasta el uso de los métodos de construcción, nuestras consultas y manejo posterior de sus resultados se ha simplificado mucho más. Una vez incorporadas al lenguaje las bondades de LINQ, estaba claro que el siguiente paso a dar por parte del equipo de desarrollo de EF era el de posibilitarnos usar el lenguaje de consultas integrado sobre nuestros objetos `ObjectSet<TEntity>`, en realidad a su subclase `ObjectQuery<TEntity>`.

Si usted revisa de nuevo el fragmento en el que se presentaba la estructura de la clase `ObjectSet<TEntity>`, podrá observar como ésta implementa la interfaz `IQueryable<T>` (y más aún, también `IOrderedQueryable<T>`, heredera de la anterior), y por lo tanto disponen de todo lo necesario para que se puedan aplicar consultas LINQ sobre los objetos de ese tipo. Esta posibilidad no solamente mejorará nuestra forma de trabajar con los modelos conceptuales, sino que además nos permitirá expresar las consultas de una forma fuertemente tipada.Con ello dejaremos de escribir consultas entrecomilladas, sobre las que el compilador no puede inferir nada, quedando la detección de los errores para tiempo de ejecución; y pasaremos a escribir consultas directamente en C#. Estas consultas sí que podrá comprobarlas el compilador, comunicándonos los errores en ellas durante la fase de desarrollo. Por supuesto, otra de las ventajas que nos proporciona LINQ es el hecho de que nuestra curva de aprendizaje se verá reducida considerablemente, y no tendremos que aprender un nuevo lenguaje como Entity SQL, ya que dispondremos de los mismos elementos con los que consultamos colecciones de objetos, XML u otras fuentes a partir de la versión 3.0 del lenguaje.

Durante las siguientes líneas, iremos viendo los distintos métodos soportados dentro de la implementación del proveedor LINQ sobre `ObjectSet<TEntity>` / `ObjectQuery<Tentity>`, conocida como **LINQ to Entities**, así como ejemplos de uso de

muchos de ellos. Para estos ejemplos partiremos de un modelo conceptual, **ef4_ex_26**, que incluya las entidades `Customer`, `Order` y `OrderDetails`, usadas justamente en los ejemplos anteriores.

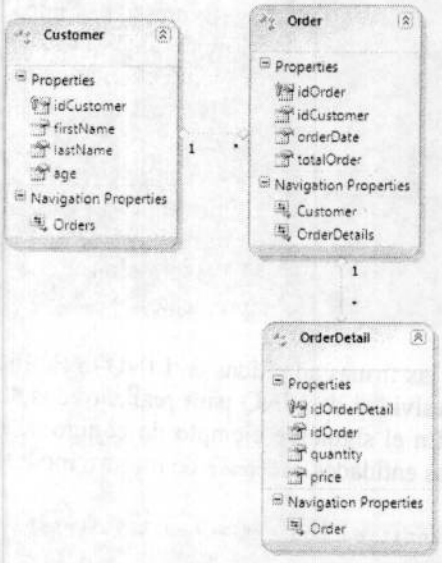


Figura 4.10.- Modelo simple para los ejemplos

2.4.1.- Métodos de proyección y restricción

Las proyecciones se refieren a la transformación de elementos de un conjunto de resultados a una forma deseada. Una proyección es, por ejemplo, recuperar solo algunas de las propiedades de un conjunto de resultados, añadir a un conjunto una propiedad que realiza una operación matemática sobre algunas de las demás propiedades, etc. Las restricciones, por otra parte, tienen que ver con la selección solo de aquellas entidades que cumplen con una condición especificada.

De los distintos métodos y firmas de los mismos dentro de los operadores de consulta de LINQ, LINQ to Entities da soporte a algunas de las firmas de los métodos `Select`, `SelectMany` y `Where`, tal y como se muestra en la siguiente tabla.

Tabla 2.- Métodos de proyección y restricción

Método	Firma
SELECT	IQueryable<TResult> Select<TSource, TResult>(this IQueryable<TSource> source, Expression<Func<TSource, TResult>>

	selector)
SELECTMANY	IQueryable<TResult> SelectMany<TSource, TResult>(this IQueryable<TSource> source, Expression<Func<TSource, IEnumerable<TResult>>> selector)
SELECTMANY	IQueryable<TResult> SelectMany<TSource, TCollection, TResult>(this IQueryable<TSource> source, Expression<Func<TSource, IEnumerable<TCollection>>> collectionSelector, Expression<Func<TSource, TCollection, TResult>> resultSelector)
WHERE	IQueryable<TSource> Where<TSource>(this IQueryable<TSource> source, Expression<Func<TSource, bool>> predicate)

Una vez conocidas las firmas admitidas en LINQ to Entities, ya podríamos empezar a utilizar toda la expresividad de LINQ para realizar consultas sobre nuestros objetos `ObjectSet<TEntity>`. En el siguiente ejemplo de código se muestra cómo obtener un enumerable de todas las entidades `Customer` de nuestro modelo conceptual.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    IQueryable<Customer> customers = from c in
                                     context.Customers
                                     select c;

}
```

Observe aquí varios puntos similares con respecto al trabajo con los métodos de construcción de consultas. En primer lugar, siempre trabajamos sobre un contexto de entidades. Es éste, como ya hemos comentado, quien conoce la conexión de Entity Client a utilizar y la información de metadatos de nuestro modelo conceptual. En segundo lugar, los elementos sobre los que aplicamos las expresiones de consulta LINQ son nuestros objetos de consulta, nuestros `ObjectSet<TEntity>`; es esta clase la que implementa la interfaz `IQueryable<T>`, realmente la clase que da implementación es la subclase `ObjectQuery<TEntity>`.

Si modificamos un poco el fragmento de código anterior, podremos ver nuevos elementos importantes en los que nos tenemos que fijar. Para ello, crearemos ahora una consulta que únicamente nos devuelva los clientes cuya propiedad `age` sea mayor que 18.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    int age = 18;
    IQueryable<Customer> customers = from c in
                                     context.Customers
                                     where c.age > age
                                     select c;
}
```

Fíjese la sencillez de la consulta anterior y en cómo esta misma consulta no tiene ninguna diferencia con respecto a si la misma operara sobre una colección de objetos, por ejemplo. Como usted ya sabe, este azúcar sintáctico que nos proporciona el lenguaje no es más que una representación al uso del conjunto de métodos extensores asociados a `IQueryable<T>`, con lo que la consulta anterior se podría reescribir de la siguiente forma.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    int age = 18;
    IQueryable<Customer> customers = context.Customers
                                .Where(c => c.age > age)
                                .Select(c => c);
}
```

Esta forma de expresión de las consultas es muy similar al uso de los métodos de construcción de consultas, pero con una diferencia muy importante: las consultas de LINQ to Entities son fuertemente tipadas, y en ella se sustituyen completamente las cadenas de caracteres de Entity SQL por expresiones lambda que el compilador transforma en árboles de expresiones.

Por supuesto, ambas formas de construcción de consultas podrían combinarse:

```
using (EF4EXEntities context = new EF4EXEntities())
{
    int age = 18;
    IQueryable<Customer> customers = context.Customers
                            .Where("it.age>@age",new
ObjectParameter("age",age))
                            .Select(c => c);
}
```

Un hecho con el que mucha gente se sorprende es con el tipo de los resultados de las consultas de LINQ to Entities. Hasta ahora, estábamos acostumbrados a obtener nuevos objetos `ObjectQuery<TEntity>`, que después podíamos enumerar, o a obtener un enumerable de tipo `ObjectResult<TEntity>`, si la consulta era ejecutada con una llamada a su método `Execute`. La implementación de los operadores de consulta de LINQ para la interfaz `IQueryable<T>` siempre devuelve estáticamente un objeto que implementa esa interfaz. Pero tenga en cuenta que en realidad ese `IQueryable<T>` que obtenemos como resultado no es otra cosa que un objeto `ObjectQuery<TEntity>`. Puede comprobar esto realizando una conversión de un tipo al otro, como en el siguiente fragmento.

```
using (EF4EXEntities context = new EF4EXEntities())
{
     ObjectQuery<Customer> customers =
(ObjectQuery<Customer>)context.Customers
                              .Select(c => c);
}
```

En los distintos métodos de construcción de consultas con Entity SQL, vimos como disponíamos de una sobrecarga para indicar los distintos parámetros que tomaban nuestras consultas. Podría parecer que en LINQ to Entities no dispusiéramos de nada similar para poder disponer de consultas parametrizadas, un recurso muy conveniente, como vimos en el capítulo anterior. Pues bien, en LINQ to Entities cualquier consulta que utilice una variable dentro de ella para referirse al valor de una propiedad se transforma en una consulta parametrizada. Por otra parte, el uso de un valor directamente dentro de la consulta no genera una consulta parametrizada. Por ejemplo, la siguiente consulta no genera una consulta parametrizada puesto que hacemos uso del valor directamente dentro de la consulta.

```
using (EF4EXEntities context = new EF4EXEntities())
{
     IQueryable<Customer> customers = from c in
                                      context.Customers
                                      where c.age > 18
                                      select c;
}
```

Sin embargo, si la refactorizamos como en el siguientes listado, entonces sí que se generará una consulta parametrizada.

```
using (EF4EXEntities context = new EF4EXEntities())
{
     int age = 18;
     IQueryable<Customer> customers = from c in
                                      context.Customers
                                      where c.age > age
                                      select c;
}
```

En muchas ocasiones, si no en la mayoría de ellas, el resultado de la consulta necesita ser almacenado en una estructura determinada, para poder transportarlo o procesarlo posteriormente; razón por la que una vez obtenido un objeto `ObjectQuery<T>` para una consulta determinada, sus resultados suelen ser transformados en listas, enumerados, vectores de elementos, diccionarios etc. Para facilitarnos este trabajo, LINQ nos proporciona distintos métodos como `ToList`, `AsEnumerable`, `ToArray` o `ToLookup`, entre otros, que automáticamente realizan la enumeración de los resultados y nos devuelven una lista, diccionario o vector con los elementos que lo forman. En el siguiente ejemplo vemos cómo utilizar el método `ToList` para obtener la lista de productos.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    List<Customer> customers = context.Customers.ToList();
}
```

Nota: Recuerde que las consultas son ejecutadas contra la base de datos subyacente cuando las mismas se enumeran, o cuando se llama directamente al método Execute de las mismas. Los métodos como ToList, ToLookup, etc... enumeran los resultados de una consulta, por lo que esta es ejecutada en ese preciso instante.

2.4.2.- Métodos de encuentro

A pesar de disponer de las propiedades de navegación, gracias a las cuales la navegación por las entidades es trivial, los métodos de encuentro o reunión son de vital importancia para reunir entidades que no tienen una relación establecida. Los métodos de reunión de los que disponemos son `Join` y `GroupJoin`.

Nota: Algunas de las firmas de estos dos métodos, `Join` y `GroupJoin`, no están soportadas en Linq To Entities debido a que hacen referencia a la interfaz `IEqualityComparer`, elemento que no puede ser traducido a la fuente de almacenamiento.

En la siguiente tabla podemos ver los métodos y las firmas soportadas actualmente por Linq To Entities.

Tabla 3.- Firmas soportadas para Join y GroupJoin

Método	Firma
Join	IQueryable<TResult> Join<TOuter, TInner, TKey, TResult>(this IQueryable<TOuter> outer, IEnumerable<TInner> inner, Expression<Func<TOuter, TKey>> outerKeySelector, Expression<Func<TInner, TKey>> innerKeySelector, Expression<Func<TOuter, TInner, TResult>> resultSelector)
GroupJoin	IQueryable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(

```
    this IQueryable<TOuter> outer,
    IEnumerable<TInner> inner,
    Expression<Func<TOuter,      TKey>>
outerKeySelector,
    Expression<Func<TInner,      TKey>>
innerKeySelector,
    Expression<Func<TOuter,
IEnumerable<TInner>, TResult>>
    resultSelector
    )
```

A continuación mostraremos un ejemplo del método join, **ef4_ex_27**, mediante el cual obtendremos una unión de nuestras entidades `Order` y `OrderDetail` representada en un tipo anónimo.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    var result = (from o in context.Orders
                  join od in context.OrderDetails
                  on o.idOrder equals od.idOrder
                  select new { Id = o.idOrder, Total = o.totalOrder
,Price = od.price });
}
```

2.4.3.- Métodos de partición

La mayoría de las firmas y métodos de partición están soportados dentro de LINQ to Entities, exceptuando los que hacen uso de `IEqualityComparer`, como podemos ver en la siguiente tabla.

Tabla 4.- Métodos de partición

Método	Firma

All	`bool All<TSource>(`
	`this IQueryable<TSource> source,`
	`Expression<Func<TSource, bool>>`
	`predicate)`

Any	`bool Any<TSource>(` `this IQueryable<TSource> source)`

Any	`bool Any<TSource>(`
	`this IQueryable<TSource> source,`
	`Expression<Func<TSource, bool>>`
	`predicate)`

Distinct	`IQueryable<TSource>`
	`Distinct<TSource>(`
	`this IQueryable<TSource> source)`

Except	`IQueryable<TSource> Except<TSource>(`
	`this IQueryable<TSource> source1,`
	`IEnumerable<TSource> source2)`

Intersect	`IQueryable<TSource>` `Intersect<TSource>(`

Union	this IQueryable<TSource> source1, IEnumerable<TSource> source2) IQueryable<TSource> Union<TSource>(this IQueryable<TSource> source1, IEnumerable<TSource> source2)

Como usted bien sabrá, ninguno de estos métodos tiene representación en la sintaxis de consultas integradas, por lo que todos ellos deben invocarse de la forma tradicional, mediante sintaxis de llamada, al igual que ocurre con los métodos de construcción de consultas. En el siguiente ejemplo mostraremos como realizar una unión de los elementos de tipo `Order` cuya propiedad `TotalOrder` se encuentre entre 2000-3000 y 4000-5000.

```
using (EF4EXEntities context = new EF4EXEntities())
{
     IQueryable<Order> result = context.Orders
                          .Where(o=>o.totalOrder > 1000
                               && o.totalOrder <2000)
                          .Union(
context.Orders.Where(o=>o.totalOrder > 4000 && o.totalOrder < 5000));

}
```

2.4.4.- Métodos de ordenación

La ordenación de consultas es otra de las necesidades más habituales cuando hablamos del trabajo con cualquier fuente de datos. LINQ to Entities soporta la mayoría de los métodos de ordenación previstos para los proveedores de LINQ. En este caso debemos de constatar que la ordenación se realiza directamente sobre la fuente de datos, a diferencia de cuando ordenamos una colección en memoria, acción que se realiza sobre tipos del CLR.Es por eso que una misma consulta de LINQ to Entities pueda diferir dependiendo de las opciones establecidas en la fuente de datos subyacente, como si ésta es sensible a mayúsculas y minúsculas, ordenaciones *Kanji* o cómo se ordenan los valores nulos.

En la siguiente tabla se listan los métodos y firmas soportados en LINQ to Entities para la ordenación de consultas.

Tabla 5.- Métodos de ordenación soportados

Método	Firma
OrderBy	IOrderedQueryable<TSource> OrderBy<TSource, TKey>(this IQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector)
OrderByDescending	IOrderedQueryable<TSource> OrderByDescending<TSource, TKey>(this IQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector)
ThenBy	IOrderedQueryable<TSource>

ThenByDescending	ThenBy<TSource, TKey>(this IOrderedQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector) IOrderedQueryable<TSource> ThenByDescending<TSource, TKey>(this IOrderedQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector)

En el siguiente ejemplo, **ef4_ex_28**, mostraremos como ordenar nuestra entidad `Customer` en función del valor de la propiedad `firstName`.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    IQueryable<Customer> result = from c in
                                    context.Customers
                                  orderby c.firstName
                                  select c;

}
```

2.4.5.- Métodos de agrupación

La mayoría de firmas del método de agregación de LINQ, `GroupBy`, están soportados dentro de la implementación de LINQ to Entities. Su uso es muy similar al que hemos estudiado para el método de construcción de consultas del mismo nombre, con la diferencia esencial de que ahora la clave y los elementos agrupados se especifican mediante código y no como cadenas de caracteres eSQL.

Tabla 6.- Métodos de agrupación soportados

Método	Firma
GroupBy	IQueryable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(this IQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector)
GroupBy	IQueryable<IGrouping<TKey, TElement>> GroupBy<TSource, TKey, TElement>(this IQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector, Expression<Func<TSource, TElement>> elementSelector)
GroupBy	IQueryable<TResult> GroupBy<TSource, TKey, TResult>(this IQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector, Expression<Func<TKey, IEnumerable<TSource>, TResult>>

GroupBy	resultSelector) IQueryable<TResult> GroupBy<TSource, TKey, TElement, TResult>(this IQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector, Expression<Func<TSource, TElement>> elementSelector, Expression<Func<TKey, IEnumerable<TElement>, TResult>> resultSelector)

Si quisiéramos, por ejemplo, mostrar los grupos de los distintos pedidos agrupados por su fecha, podríamos hacer algo como lo mostrado en el siguiente fragmento, **code\EF4_EX_29**.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    IEnumerable<IGrouping<DateTime,Order>> result = (from o
                                in context.Orders
                                group o by o.orderDate);
}
```

2.4.6.- Métodos de agregados

Una operación de agregación no es más que un cálculo sobre valores simples dentro de una colección de valores, como por ejemplo la obtención de la media de los importes de los pedidos realizados, del máximo tiempo de demora en la entrega de un determinado conjunto de pedidos, etc. Al igual que ocurre con los métodos de ordenación, los métodos de agregación usan la semántica propia de la base de datos subyacente, de tal forma que un método agregado como `Sum` no dará ningún error si sumara valores nulos contenidos en la base de datos.

Nota: Los métodos agregados que involucren cálculos sobre una secuencia, como *Sum* o *Average*, son ejecutados por el motor de base de datos, por lo que durante la conversión de tipos de la base de datos a los del CLR, podría ocurrir una merma en la precisión.

Los métodos de agregación y firmas soportados se muestran en la siguiente tabla:

Tabla 7.- Métodos de agregación soportados

Método	Firma
Average	decimal Average(this IQueryable<decimal> source)
Average	double Average(this IQueryable<double> source)
Average	double Average(this IQueryable<int> source)
Average	double Average(this IQueryable<long> source)

Average	`Nullable<decimal> Average(this IQueryable<Nullable<decimal>> source)`
Average	`Nullable<double> Average(this IQueryable<Nullable<double>> source)`
Average	`Nullable<double> Average(this IQueryable<Nullable<int>> source)`
Average	`Nullable<double> Average(this IQueryable<Nullable<long>> source)`
Average	`Nullable<float> Average(this IQueryable<Nullable<float>> source)`
Average	`float Average(this IQueryable<float> source)`
Average	`int Count<TSource>(this IQueryable<TSource> source)`
LongCount	`long LongCount<TSource>(this IQueryable<TSource> source)`
Max	`TSource Max<TSource>(this IQueryable<TSource> source)`
Min	`TSource Min<TSource>(this IQueryable<TSource> source)`
Sum	`decimal Sum(this IQueryable<decimal> source)`
Sum	`double Sum(this IQueryable<double> source)`
Sum	`int Sum(this IQueryable<int> source)`
Sum	`long Sum(this IQueryable<long> source)`
Sum	`Nullable<decimal> Sum(this IQueryable<Nullable<decimal>> source)`
Sum	`Nullable<double> Sum(this IQueryable<Nullable<double>> source)`
Sum	`Nullable<int> Sum(this IQueryable<Nullable<int>> source)`
Sum	`Nullable<long> Sum(this IQueryable<Nullable<long>> source)`
Sum	`Nullable<float> Sum(this IQueryable<Nullable<float>> source)`
Sum	`float Sum(this IQueryable<float> source)`

A continuación se presentan algunos ejemplos de uso de estos métodos de agregación, **code\ef4_ex_30.**

```
using (EF4EXEntities context = new EF4EXEntities())
{
      decimal resultSum = context.Orders.Sum(o => o.totalOrder);
      decimal resultAvg = context.Orders.Average(o => o.totalOrder);
      decimal resultMin = context.Orders.Min(o => o.totalOrder);
}
```

El comportamiento en función de si los datos contienen valores nulos o no puede variar en función del método de agregación de que se trate. La siguiente tabla describe estos comportamientos para los distintos métodos de agregación presentes en LINQ to Entities.

Tabla 8.- Comportamientos de los métodos de agregación

Método	Sin datos	Todos nulos	Algunos nulos	Sin nulos
Count	El resultado es 0	El resultado es la cantidad de valores nulos encontrados	El resultado es la cantidad de de valores nulos y no nulos encontrados	Devuelve la cantidad de elementos en la secuencia
Sum	El resultado es null	El resultado es null	El resultado es la suma de los valores no nulos	El resultado es la suma de todos los valores no nulos en la secuencia
Max	El resultado es null	El resultado es null	El resultado es el máximo de los valores no nulos	El resultado es el máximo de los elementos en la secuencia
Min	El resultado es null	El resultado es null	El resultado es el mínimo de los valores no nulos	El resultado es el mínimo de los elementos en la secuencia

2.4.7.- Métodos de elementos y paginación

Al contrario que en otros casos, en lo relativo a los métodos de elementos (que permiten obtener un único elemento) o de paginación (que producen entidades correspondientes a un subconjunto contiguo de filas relacionales), solamente unos cuantos operadores de consulta estándar de LINQ están soportados en LINQ to

Entities. Debido a la naturaleza de la fuente de almacenamiento, una base de datos relacional, métodos como `Single` o `Last` son difíciles de implementar.

Aún así, el trabajo de paginación de consultas, al igual que con los métodos de construcción, se simplifica enormemente con respecto a como lo hacíamos de forma tradicional con ADO.NET. Si en los métodos de construcción disponíamos de los métodos `Skip` y `Top` que nos hacían la vida muy simple, ahora en LINQ to Entities disponemos de los métodos `Skip` y `Take`.

Tabla 9.- Métodos soportados de elementos y paginación

Firma	Método
First	`TSource First<TSource>(this IQueryable<TSource> source)`
First	`TSource First<TSource>(this IQueryable<TSource> source, Expression<Func<TSource, bool>> predicate)`
FirstOrDefault	`TSource FirstOrDefault<TSource>(this IQueryable<TSource> source)`
FirstOrDefault	`TSource FirstOrDefault<TSource>(this IQueryable<TSource> source, Expression<Func<TSource, bool>> predicate)`
Skip	`IQueryable<TSource> Skip<TSource>(this IQueryable<TSource> source, int count)`
Take	`IQueryable<TSource> Take<TSource>(this IQueryable<TSource> source, int count)`

En el siguiente ejemplo, **code\ef4_ex_31**, podemos ver un fragmento de código que obtiene la segunda de las páginas, con elementos tomados de diez en diez, del conjunto de entidades de tipo `Customer`.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    int pageIndex = 1;
    int pageCount = 10;
    List<Customer> customers = context.Customers
                               .OrderBy(c => c.firstName)
                               .Skip(pageIndex*pageCount)
                               .Take(pageCount)
                               .ToList();

}
```

Nota: Le recomendamos desde aquí revisar el apéndice de tips de rendimiento y el caso de las consultas con paginación.

2.5.- Gestión del estado

Cuando empezamos en este capítulo hablando sobre la definición de nuestro contexto de trabajo, dejamos claro como éstos son la implementación del patrón **Unity Of Work**, definido por Martin Fowler en PoEAA. La responsabilidad principal de este patrón es la de "*mantener una lista de objetos afectados en una transacción y coordinar la escritura de los cambios así como la resolución de problemas de concurrencia.*". Hasta ahora, hemos visto los elementos `ObjectQuery<TEntity>` y `ObjectSet<TEntity>` así como los diferentes tipos de entidades que podemos crear a partir de nuestros modelos de dominio, pero nada acerca de cómo hacer un seguimiento de las operaciones que sobre las entidades consultadas se realizan, tal y como dicta el patrón. Para esta tarea, **ADO.NET Entity Framework 4.1** dispone de una pieza denominada ObjectStateManager, pieza responsable de almacenar los estados de las distintas entidades, así como de mantener la consistencia de los grafos formados por las distintas entidades obtenidas por medio de nuestro contexto de trabajo.

2.5.1.- ObjectStateEntry

Cada vez que una consulta es ejecutada dentro de un contexto de trabajo, por defecto, para cada una de las entidades obtenidas se crea una estructura de datos denominada `ObjectStateEntry`. Esta clase, cuya definición podemos ver en el siguiente listado, permite a EF 4.1 almacenar distintos elementos de vital importancia asociados a la entidad, como por ejemplo, la referencia a ella, los valores originales que tenía cuando fue consultada o el estado actual de la entidad, entre otros.

```
public abstract class ObjectStateEntry
: IEntityStateEntry,
  IEntityChangeTracker
{

        public abstract CurrentValueRecord CurrentValues { get; }

        public abstract object Entity { get; }

        public abstract EntityKey EntityKey { get; internal set; }

        public EntitySetBase EntitySet { get; }

        public abstract bool IsRelationship { get; }

        public ObjectStateManager ObjectStateManager { get; }

        public abstract DbDataRecord OriginalValues { get; }

        public abstract RelationshipManager RelationshipManager {
get; }

        public EntityState State { get; internal set; }
```

```
        public abstract void AcceptChanges();

        public abstract void ApplyCurrentValues(object
currentEntity);

        public abstract void ApplyOriginalValues(object
originalEntity);

        public abstract void ChangeState(EntityState state);

        public abstract void Delete();

        public abstract IEnumerable<string> GetModifiedProperties();

        public abstract OriginalValueRecord
GetUpdatableOriginalValues();

        public abstract void SetModified();

        public abstract void SetModifiedProperty(string
propertyName);
}
```

El conjunto de estructuras de este tipo almacenadas en un contexto es accesible a través de una propiedad de los contextos de trabajo llamada `ObjectStateManager`. Su tipo, del mismo nombre, nos proporciona distintos métodos con el fin de obtener y manejar los distintos `ObjectStateEntry` almacenados.

```
public class ObjectStateManager : IEntityStateManager
{
   // ...

   public IEnumerable<ObjectStateEntry>
      GetObjectStateEntries(EntityState state);

   public ObjectStateEntry GetObjectStateEntry(
      EntityKey key);
   public ObjectStateEntry GetObjectStateEntry(
      object entity);

   public bool TryGetObjectStateEntry(EntityKey key,
      out ObjectStateEntry entry);
   public bool TryGetObjectStateEntry(object entity,
      out ObjectStateEntry entry);

   // ...
}
```

Con la finalidad de ir comprendiendo mejor el funcionamiento de la clase `ObjectStateManager` y cómo se crean los objetos `ObjectStateEntry`, presentaremos distintos ejemplos de consulta y trataremos de mostrar de una forma gráfica los distintos elementos `ObjectStateEntry` que se van creando y almacenando en esta importante pieza de EF. Los ejemplos se basarán en el mismo modelo de ejemplo con

el que hemos ido tratando anteriormente con nuestras entidades `Customer`, `Order` y `OrderDetails`.

Empezaremos partiendo del siguiente ejemplo de código, en el que se obtiene el primer cliente que cumple el criterio de edad mayor que dieciocho.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    int age = 18;
    Customer customer = (from c in context.Customers
                         where c.age > age
                         select c).FirstOrDefault();
}
```

ADO.NET Entity Framework, además de las entidades resultantes de la consulta (en este caso una), para cada entidad obtenida crea un nuevo elemento `ObjectStateEntry`, en el que incluye la propiedad `EntityKey` de la entidad, una referencia al objeto materializado, el estado de la entidad y dos propiedades para realizar el seguimiento de los cambios de estado; concretamente, sus valores originales y los valores de las propiedades que la entidad tiene en cada momento, sus valores actuales.

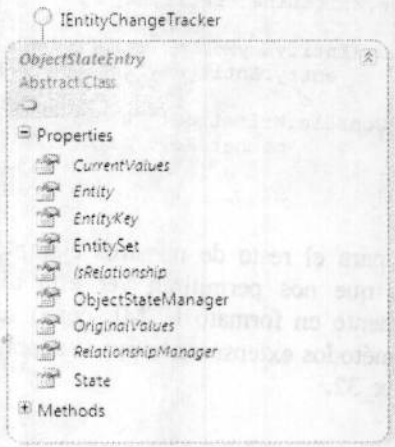


Figura 4.11.- ObjectStateEntry

El valor de la propiedad `State` nos permite conocer el estado de la entidad dentro del contexto. Este valor puede ser alguno de los contenidos en la enumeración `EntityState`.

```
public enum EntityState
{
    Detached = 1,
    Unchanged = 2,
    Added = 4,
    Deleted = 8,
    Modified = 16,
}
```

El valor `Unchanged` es el valor por defecto de todas las estructuras `ObjectStateEntry` creadas a partir de una consulta. El resto de valores indican el estado en el que se encuentran las entidades asignadas a cada `ObjectStateEntry`, y su asignación se hace generalmente de forma automática. Así, el valor `Deleted` significa que la entidad asociada ha sido marcada para eliminarse, el valor `Modified` indica que se ha modificado alguna propiedad de la entidad, y el valor `Added` que la entidad asociada a esta estructura no es una entidad consultada por el modelo, sino que es una entidad creada desde el código y lista para que se realice una inserción de la misma en la base de datos subyacente.

Por su parte, los registros `OriginalValues` y `CurrentValues` nos permiten almacenar los valores originales y actuales de la entidad asociada a esta estructura, respectivamente. Estos valores se corresponden con el tipo `IExtendedDataRecord`, visto en el capítulo sobre Entity Client, que a su vez almacena un `EntityType` o un `AssociationType`.

La propiedad `Entity` de la estructura `ObjectStateEntry` es una referencia al objeto al que representa la entidad consultada, y `EntityKey` es la estructura que representa la clave de dicha entidad; aquello que permite distinguirla unívocamente del resto.

En el siguiente ejemplo de código, vemos cómo obtener parte de la información de la estructura `ObjectStateEntry` creada a partir de la consulta anterior.

```
static void Dump(ObjectStateEntry entry)
{
     Console.WriteLine("Estado:{0}", entry.State);
     Console.WriteLine("EntityKey");

     foreach (EntityKeyMember member in
               entry.EntityKey.EntityKeyValues)
     {
          Console.WriteLine("\t Clave:{0} Valor:{1}",
                  member.Key, member.Value);
     }
}
```

Por simplicidad para el resto de nuestros ejemplos, hemos creado una serie de métodos extensores que nos permitirán ver esta información de una forma más amigable, concretamente en formato HTML como se aprecia en la figura 4.12. La definición de estos métodos extensores la puede encontrar en el mismo código de este ejemplo, **code\ef4_ex_32.**

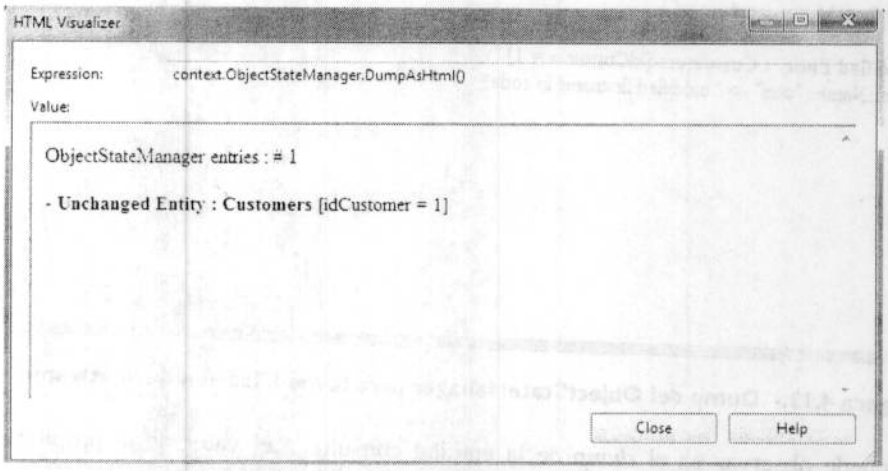


Figura 4.12.- Visualizador HTML de nuestras ObjectStateEntry

Como comentamos en un párrafo anterior, las estructuras `CurrentValues` y `OriginalValues` de nuestros elementos `ObjectStateEntry` son elementos del tipo `IExtendedDataRecord`. Estos elementos son de vital importancia en los procesos de eliminación y actualización de entidades puesto que gracias a ellos el sistema, no solamente es capaz de saber los valores que tienen las propiedades de una determinada entidad en un momento dado, sino que además es capaz de conocer los valores originales que tenía esa entidad cuando fue consultada. Esto último es esencial para implementar un mecanismo de concurrencia optimista. Cada vez que una entidad es consultada (obtenida como resultado de una consulta), dentro de la estructura `OriginalValues` se asigna el `EntityType` correspondiente a la entidad con los valores obtenidos en la consulta; estos mismos valores se establecen igualmente en la propiedad `CurrentValues`. A partir de ahí, si se modifica alguna de las propiedades de la entidad estos nuevos valores se van almacenando en `CurrentValues`, mientras que la propiedad `OriginalValues` se mantiene intacta.

En el siguiente listado vemos un ejemplo de esto último, en el que obtenemos una entidad `Customer`, en la que modificamos el valor de una propiedad y posteriormente mostramos en nuestro visualizador del `ObjectStateManager` el valor que esa propiedad contiene en `CurrentValues` y en `OriginalValues`, figura 4.13.

```
using (EF4EXEntities context = new EF4EXEntities())
{
     int age = 18;
     Customer customer = (from c in context.Customers
                          where c.age > age
                          select c).FirstOrDefault();

      customer.firstName = " modified firstname in code!";

      //add breakpoint in last line and check
      //context.ObjectStateManager.DumpAsHtml() with Html
      //visualizer
}
```

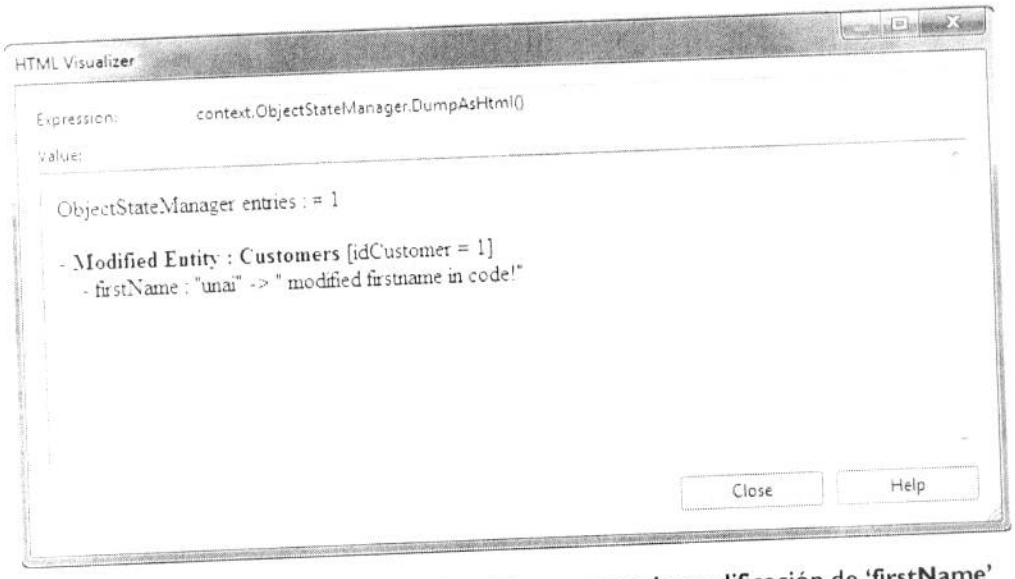


Figura 4.13.- Dump del ObjectStateManager para la modificación de 'firstName'

Como puede observar en el *dump* de la entidad consultada el valor de la propiedad firstName está modificado pasando de *Unai* a *"modified firstName in code!"*

Esta capacidad que nos proporciona EF para realizar el seguimiento de las entidades (valores originales y valores actuales) así como el estado que presentan las entidades es configurable, como cabría esperar, para los distintos objetos de consulta `ObjectQuery<TEntity>` presentes en un modelo conceptual. Esto es bien lógico puesto que esta característica implica un mayor consumo de recursos. Piense que mantener el estado de los valores originales implica un mayor consumo de memoria. Seguramente, si ha probado a ejecutar algunos de los ejemplos de las secciones anteriores, ha visto como estos objetos de consulta nos permiten establecer una propiedad denominada `MergeOption`, cuyo valor corresponde al de un tipo enumerado del mismo nombre. Esta enumeración, cuyos posibles valores podemos ver en la siguiente tabla, nos permite especificar si las entidades consultadas deben tener su correspondiente estructura `ObjectStateEntry` o no y cómo combinar las distintas estructuras creadas.

Tabla 6.- Haga clic aquí para escribir texto.

Nombre	Significado
AppendOnly	Los objetos que ya existen en el contexto no son cargados de la fuente de almacenamiento. Este es el comportamiento por defecto.
NoTracking	Las entidades consultadas no reciben seguimiento por parte de la infraestructura de ObjectStateManager. Las ObjectStateEntries están en estado Detached, no existen en realidad.
OverwriteChanges	Los objetos siempre se cargan de la fuente de datos y los valores almacenados en la estructura

	`ObjectStateEntry` son reemplazados por los obtenidos de la nueva consulta.
PreserveChanges	Cuando un objeto tiene su representante `ObjectStateEntry`, este objeto actualiza los elementos contenidos en su estructura OriginalValues pero mantiene intactos los cambios. En realidad esta opción es la más compleja porque depende de varias situaciones. En EF 4.1 además se ha cambiado su comportamiento, lo veremos en detalle.

El valor por defecto para los objetos de consulta y para los distintos métodos de carga (explícita con el método `Load` o implícita si tenemos activado `Lazy Loading`) es siempre `AppendOnly`. Éste especifica que únicamente se introduzcan en el contexto las estructuras `ObjectStateEntry` que no estuvieran ya presentes en el mismo al hacer una consulta, con lo cual, aunque hubiera valores modificados en la base de datos para esos elementos, los originales se conservarían.

Un valor importante de esta enumeración es `NoTracking`. Este valor indica que no se realizará seguimiento para los objetos consultados. El estado de estas entidades será siempre `Detached`. Como veremos un poco más adelante, esto no solamente tendrá influencia en el seguimiento de los objetos, sino que también tendrá influencia en la construcción de las relaciones entre ellos.

Los dos últimos valores de esta enumeración, `OverwriteChanges` y `PreserveChanges`, tienen relación con el tratamiento de las estructuras `ObjectStateEntry` cuando éstas ya están presentes dentro del contexto y debieran ser creadas nuevamente como resultado de otra consulta. `OverwriteChanges` es la versión "*el último gana*"; es decir, en caso de que este valor esté asociado a un objeto de consulta y tengamos un `ObjectStateEntry` almacenado en el contexto de trabajo, si otra consulta implicara obtener el mismo elemento la estructura `ObjectStateEntry` existente se reemplazaría por la nueva, perdiéndose los cambios realizados sobre la anterior. `PreserveChanges`, por el contrario, conserva los cambios realizados y no los sobrescribe con los obtenidos por medio de la nueva consulta. Ilustraremos en los siguientes ejemplos el uso de Overwrite y PreserveChanges con el fin de que el lector entienda en más detalle el proceso.

En el primer ejemplo, a continuación en el siguiente párrafo, vemos como consultar una entidad `Customer` y modificar su propiedad `firstName`, para después esperar una entrada de usuario. Durante esta espera aprovecharemos para actualizar la columna `firstName` directamente en nuestra base de datos. Posteriormente refrescaremos desde la base de datos la consulta con el fin de notar el comportamiento de esta opción.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    context.Customers.MergeOption = MergeOption.OverwriteChanges;
    int age = 18;
    Customer customer = (from c in context.Customers
                         where c.age > age
                         select c).FirstOrDefault();
```

```
        customer.firstName = "Modificado en aplicación";

        //esperamos para que se modifique la bd directamente
        Console.WriteLine("Realizamos la espera");
        Console.ReadLine();

        customer = (from c in context.Customers
                    where c.age > age
                    select c).FirstOrDefault();

          //add breakpoint in last line and check
          //context.ObjectStateManager.DumpAsHtml() with Html
          //visualizer
    }
```

Como puede observar en la figura 4.14, una vez que hemos obtenido la entidad y modificada su propiedad, si revisamos las entradas en nuestro ObjectStateManager veremos como disponemos de una entrada con estado en "Modified" y el cambio de la propiedad `firstName` de *"unai"* a *"modificado en aplicación"*.

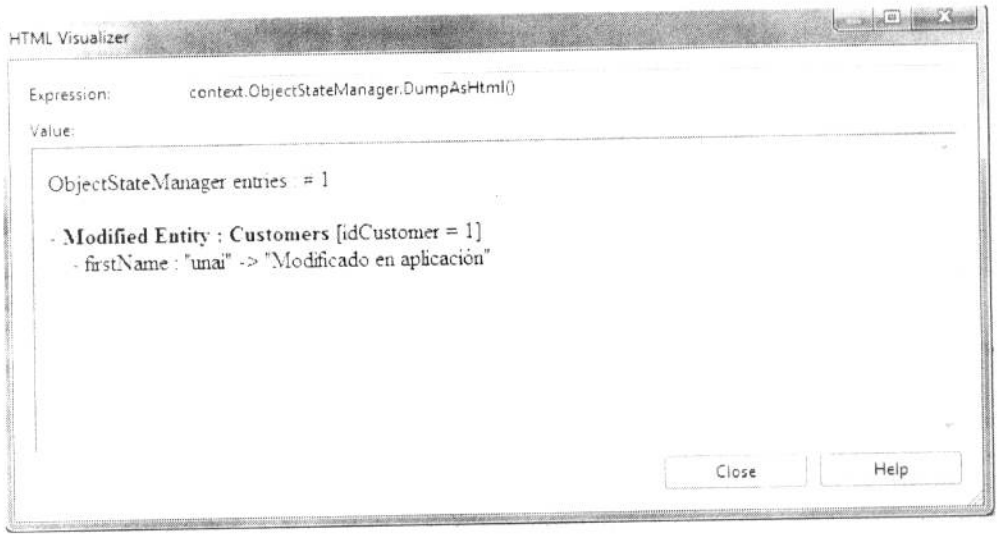


Figura 4.14.- ObjectStateManager antes del refresco

Una vez hecha la modificación en la base de datos directamente, si quitamos la espera de nuestra aplicación, `Console.ReadLine()`, y volvemos a refrescar el dato de la entidad `Customer` podremos observar como con la opción `OverwriteChanges` no solamente los datos originales se han refrescado, sino que también los actuales y, por lo tanto, los cambios realizados se han perdido. Por supuesto, el estado de la `ObjectStateEntry` asociada, ha pasado otra vez al estado `Unchanged`, Figura 4.15.

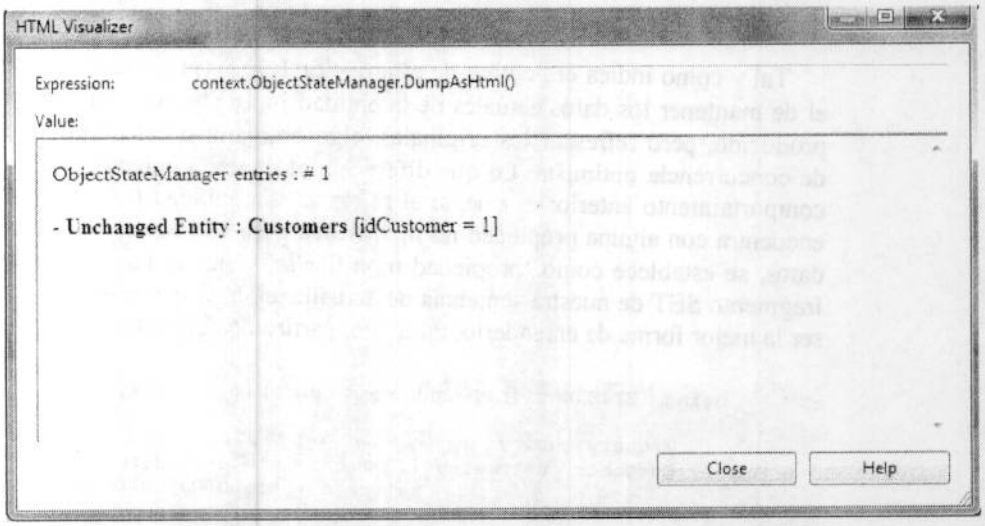


Figura 4.15.- ObjectStateManager después del refresco

La figura 4.16 muestra los datos de la entidad, con el nuevo valor de la propiedad puesto en la base de datos.

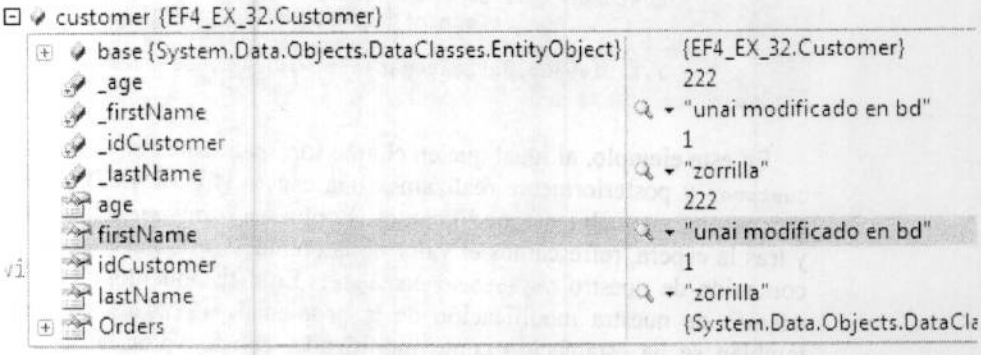


Figura 4.16.- Revisión de los valores de la entidad

El uso de `PreserveChanges` es algo más complicado, realmente porque puede tener una dualidad de comportamiento: tal y como funcionaba en Entity Framework 3.5 y el nuevo introducido en esta versión, ADO.NET EF 4.1. Para establecer uno u otro comportamiento puede modificar el valor de la propiedad `UseLegacyPreserveChangesBehavior` en las propiedades de un contexto de trabajo, a continuación en el siguiente fragmento:

```
using (EF4EXEntities context = new EF4EXEntities())
{
    //use ef 3.5 behavior

    context.ContextOptions.UseLegacyPreserveChangesBehavior = true;

   //use ef 4.1 behavior

    context.ContextOptions.UseLegacyPreserveChangesBehavior = false;
}
```

Tal y como indica el nombre de esta opción (`MergeOption`), su principal objetivo es el de mantener los datos actuales de la entidad junto con los cambios que se hubieran producido, pero refrescar los originales, algo obligatorio para resolver un mecanismo de concurrencia optimista. Lo que diferencia al comportamiento actual de EF 4.1, del comportamiento anterior es que, si al refrescar una entidad (en estado modificado) se encuentra con alguna propiedad no modificada pero con valor distinto al de la base de datos, se establece como 'propiedad modificada' y por lo tanto se incluye dentro del fragmento SET de nuestra sentencia de actualización. Como siempre, un ejemplo suele ser la mejor forma de entenderlo. Para ello, partiremos del siguiente código.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    //query entity
    Customer customer = (from c in context.Customers
                         select c).FirstOrDefault();

    customer.firstName = "modificado en aplicación";
    Console.WriteLine("Realizamos una espera, modifique la BD");
    Console.ReadLine();

    context.Customers.MergeOption =
                  MergeOption.PreserveChanges;

    customer = (from c in context.Customers
                select c).FirstOrDefault();

    //Review ObjectStateManager with HTML Visualizer
}
```

En este ejemplo, al igual que en el anterior, realizamos una consulta de una entidad `Customer` y posteriormente realizamos una espera (figura 4.17). Mientras estamos en esta espera, manualmente modificamos la columna *lastName*. Una vez hecho el cambio y tras la espera, refrescamos el valor de la entidad *Customer* y observamos otra vez el contenido de nuestro `ObjectStateManager`. En éste podemos ver (figura 4.18) como además de nuestra modificación de la propiedad `firstName`, la propiedad `lastName` también se ha establecido como modificada, con el valor de la bd, y por lo tanto formará parte de nuestra sentencia de actualización.

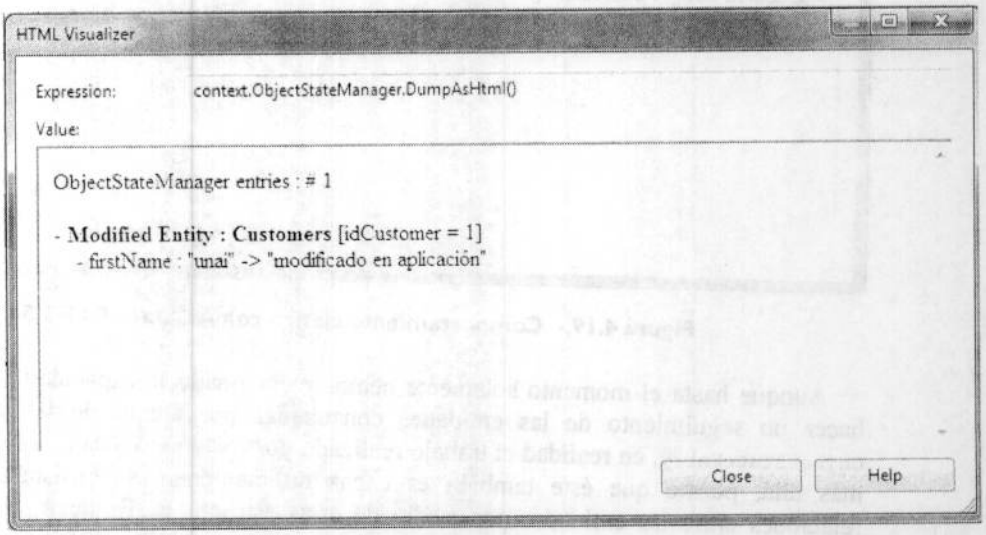


Figura 4.17.- PreserveChanges, antes del refresco

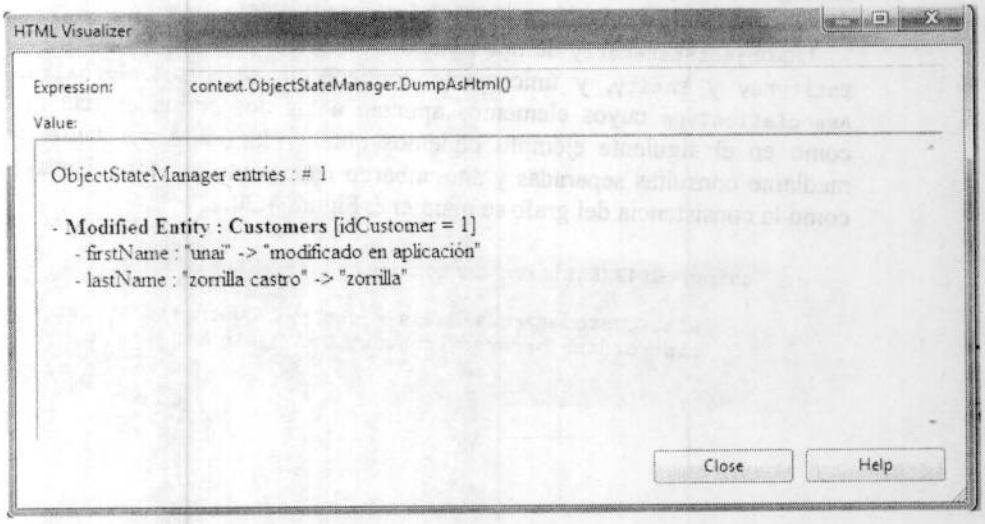


Figura 4.18.- PreserveChanges, después del refresco

Si por el contrario, hubiéramos establecido la opción de comportamiento clásica, `UseLegacyPreserveChangesBehavior`, el valor de la propiedad `lastName` no aparecería como modificado, Figura 4.19.

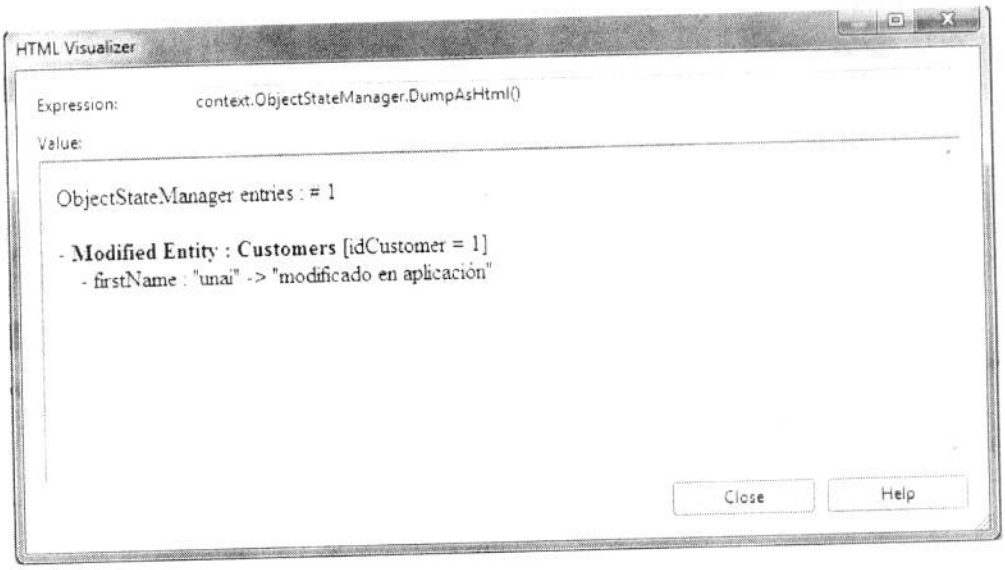


Figura 4.19.- Comportamiento clásico con ADO.NET EF 3.5

Aunque hasta el momento solamente hemos mencionado la capacidad de EF para hacer un seguimiento de las entidades consultadas por medio de las estructuras `ObjectStateEntry`, en realidad el trabajo realizado por `ObjectStateManager` va un poco más allá, puesto que éste también es capaz de mantener la consistencia de las relaciones entre las entidades por medio de estas estructuras. Es decir, las distintas estructuras `ObjectStateEntry` también se relacionan con otras estructuras, del mismo modo que una entidad se relaciona con otras entidades. Para ello, `ObjectStateManager` distingue tres tipos de estructuras `ObjectStateEntry`: las tradicionales (que acabamos de ver), las **de tipo relación** y los **stub** (representantes).

Un `ObjectStateEntry` de tipo relación no contiene ningún valor en las propiedades `EntityKey` y `Entity`, y únicamente contiene un registro `OriginalValues` con un `AssociationType` cuyos elementos apuntan a las dos entidades relacionadas. Fíjese como en el siguiente ejemplo podemos obtener las entidades *Customer* y *Orders* mediante consultas separadas y sin embargo navegando por las mismas podemos ver como la consistencia del grafo se mantiene, Figura 4.20.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    List<Customer> customers = context.Customers.ToList();
    List<Order> orders = context.Orders.ToList();
}
```

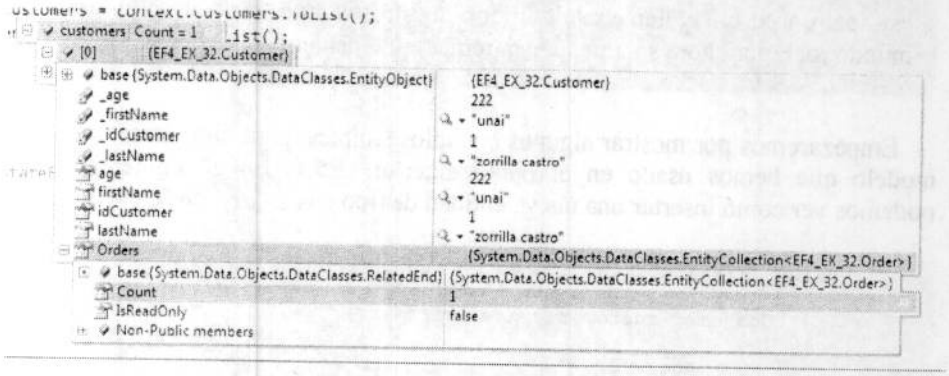


Figura 4.20.- Consistencia del grafo

2.5.2.- Inserción de entidades

Por fin, después de tantas y tantas líneas sobre la creación de modelos conceptuales y la consulta de éstos, ha llegado el momento de ver cómo se pueden realizar operaciones DML (de manipulación de datos: inserciones, actualizaciones y borrados) de elementos.

Al igual que en el caso de las consultas, una de las piezas fundamentales relacionadas con las actualizaciones es el contexto de trabajo y el `ObjectStateManager`. En la versión anterior los métodos de inserción de entidades formaban parte de los métodos de nuestros contextos de trabajo, uno por `EntitySet` disponible en nuestro modelo. En la actualidad, si bien estos métodos siguen disponibles, con la generación de código por defecto (por compatibilidad para atrás, están comentados como obsoletos), la inserción de nuevos elementos también forma parte de nuestro tipo *ObjectSet<TEntity>*, y es sobre este tipo dónde deberíamos trabajar.

```
public class ObjectSet<TEntity> : ObjectQuery<TEntity>,
IObjectSet<TEntity>, IQueryable<TEntity>, IEnumerable<TEntity>,
IQueryable, IEnumerable where TEntity : class
{

public EntitySet EntitySet { get; }


    public void AddObject(TEntity entity);

    //-- simplificado por brevedad

}
```

Puede observar como la utilización de `AddObject` es muy simple: solicitando por parámetro el objeto a insertar (del tipo definido por el `ObjectSet<TEntity>`) insertamos una instancia de una clase y no una colección más o menos desordenada de valores por medio de una sentencia **INSERT INTO**, tal y como estábamos acostumbrados a hacer con el ADO.NET tradicional.

Nota: En realidad, la idea de dotar a este nuevo tipo **ObjectSet<TEntity>** de la posibilidad de agregar (también eliminar) junto con las que ya vimos de consultar, no es más que la de dejar claro sobre este tipo la implementación de un patrón **Repository** (Martin Fowler en PoEAA). Puesto que hablaremos más en detalle de este patrón en el siguiente capítulo con nuestro ejemplo de una aplicación en el mundo real, por ahora solamente dejaremos esta mención.

Empezaremos por mostrar algunos ejemplos simples; para ello, usaremos el mismo modelo que hemos usado en el punto anterior, 2.5.1. En el siguiente fragmento podemos ver como insertar una nueva entidad de tipo `Customer`, **code\ef4_ex33.**

```
using (EF4EXEntities context = new EF4EXEntities())
{
     Customer customer = new Customer()
     {
          Age = 2,
          FirstName = "Lucia",
          LastName = "Zorrilla",
          IdCustomer = 2
     };

     context.Customers.AddObject(customer);

     context.SaveChanges();

}
```

En el código anterior nos detendremos en un par de aspectos importantes. En primer lugar podemos ver como la inserción de elementos efectivamente se basa en una llamada al método `AddObject`. La instancia de la clase a insertar, en este caso una entidad de tipo `Customer`, puede construirse normalmente mediante el uso del operador `new` (llamando a alguno de los constructores de la entidad), o por medio del uso del método estático de construcción del que todas las entidades disponen (en este caso con la plantilla de clases prescriptivas).

Estos métodos estáticos, en nuestro caso `CreateCustomer`, tienen como parámetros las propiedades de la entidad, que no pueden ser nulas. Una vez que hemos construido la entidad a insertar y llamado al método `AddObject` del `ObjectSet<TEntity>` correspondiente, el último paso es el de aplicar o procesar la inserción. Como dijimos anteriormente, los contextos de trabajo son una implementación del patrón "Unit of Work", y por ello, mientras no se realice una llamada a un método que confirme o comprometa todos los cambios realizados, estos cambios no se propagarán a la base de datos subyacente. El método que nos permite aplicar las operaciones pendientes en un contexto de trabajo se llama `SaveChanges`.

Internamente, la llamada a `AddObject` crea una nueva estructura `ObjectStateEntry` cuyo estado es `Added`, y asigna a ésta los distintos valores asociados a la nueva entidad, como su clave o los valores actuales que contiene, Figura 4.21. Cada llamada al método

`SaveChanges` implica que el contexto busque por medio del `ObjectStateManager` las estructuras marcadas con el estado `Added`, y que para cada una de ellas ejecute una sentencia de inserción con los datos correspondientes (a lo que hay que añadir las acciones asociadas a los borrados y actualizaciones, que veremos más adelante).

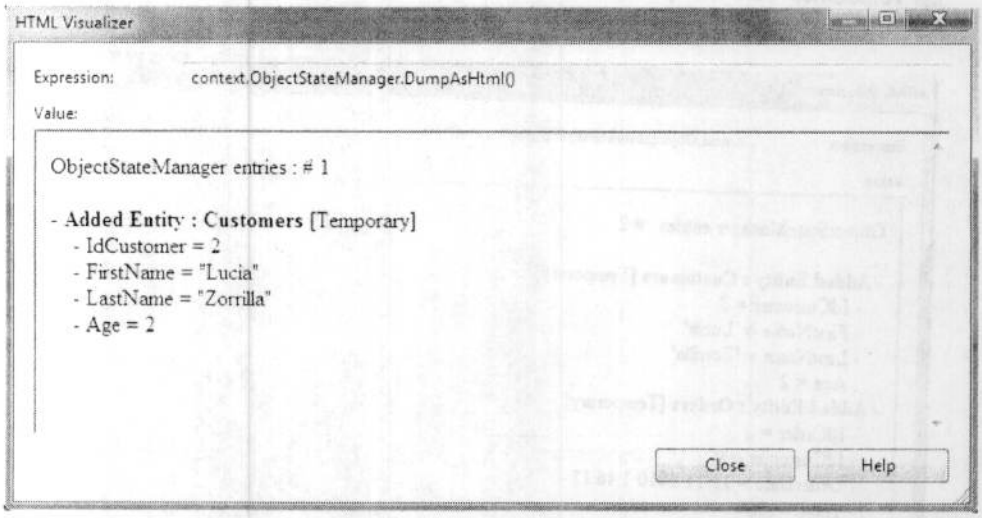


Figura 4.21.- ObjectStateManager con la inserción

El mismo ejemplo anterior lo podríamos complicar un poco más. En muchas ocasiones, nos interesará guardar un determinado grafo de elementos. Es decir, no solamente guardar una entidad en sí, sino además guardar sus relaciones con otras entidades. Imagine que desea guardar una entidad de tipo `Customer` y su relación con una entidad de tipo `Order`. Lo ideal sería no tener que realizar dos llamadas al método `AddObject`, uno por entidad, sino que con una simple llamada el sistema fuera capaz de realizar todas las operaciones de inserción necesarias. De hecho, esto está muy ligado al patrón "Aggregate" tan común en modelos de dominio, donde no se define un repositorio de consulta y modificación para todas las entidades de un modelo. Para ver la inserción de un grafo modificaremos el código del ejemplo anterior de la siguiente forma:

```
using (EF4EXEntities context = new EF4EXEntities())
{

     Customer customer = new Customer()
     {
          Age = 2,
          FirstName = "Lucia",
          LastName = "Zorrilla",
          IdCustomer = 2
      };

      customer.Orders.Add(new Order()
                          {
                              IdOrder = 2,
                              OrderDate = DateTime.UtcNow,
```

```
                Total = 1000M
            });

        context.Customers.AddObject(customer);

        context.SaveChanges();
    }
```

Fíjese ahora como el `ObjectStateManager` contiene dos elementos de tipo `ObjectStateEntry` con su estado como `Added`, figura 4.22. Fíjese también como se ha hecho uso de la propiedad de navegación entre `Customer` y `Order` para crear el grafo de las relaciones con las que el sistema sabrá qué inserciones (y en qué orden) deberá realizar en la base de datos.

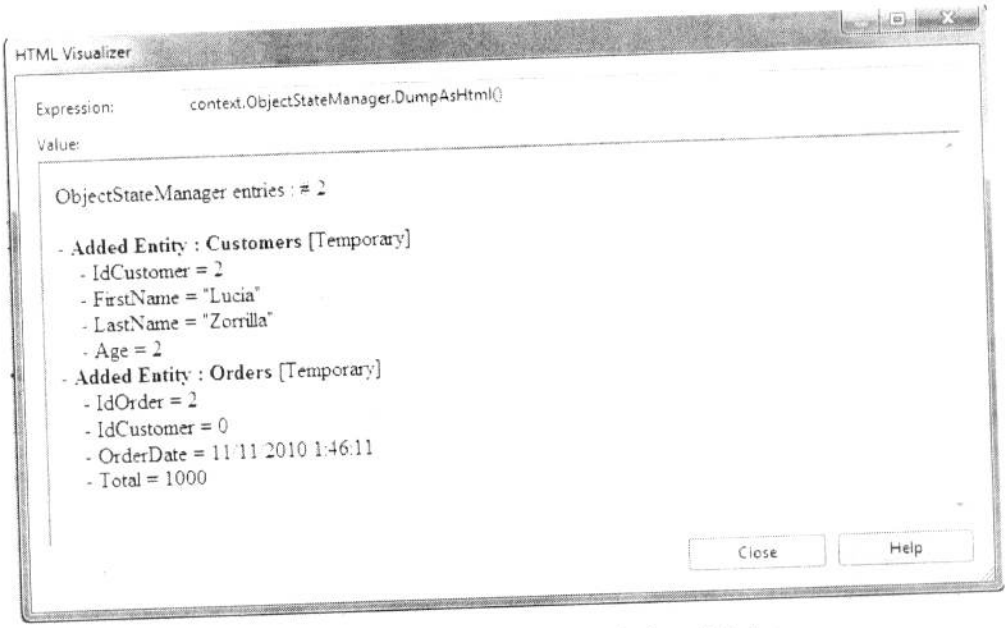


Figura 4.22.- Inserción de grafo de entidades

Nota: La resolución de claves primarias en un grafo de entidades depende del tipo de clave puesto en la entidad. Si ésta tiene el atributo de patrón de generación como identidad, EF recuperará estos valores de las secuencias para utilizarlos en posteriores inserciones.

Hay numerosos ejemplos prácticos de inserción de grafos, como por ejemplo la inserción de un pedido y todas las líneas del pedido, la inserción de un cliente y su dirección, etc. En ocasiones, según cuál sea nuestro modelo conceptual, puede que EF tenga que utilizar también varias sentencias de inserción para agregar una nueva entidad. Imagine, por ejemplo, que una entidad de un modelo EDM está mapeada a

varias tablas, con relaciones entre ellas por medio de un JOIN, o, que insertemos una entidad dentro de una jerarquía de herencia, con herencia por subtipo.

Le recomiendo desde aquí hacer ejercicios de inserción con diferentes modelos de mapeos de entidades, *Entity Spliting*, *Table Spliting*, entidades en una jerarquía etc.

2.5.3.- Eliminación de entidades

Si para insertar nuevos elementos el tipo `ObjectSet<TEntity>` nos proporciona un método llamado `AddObject`, para eliminarlos disponemos de un método denominado `DeleteObject`, cuyo uso es tan simple como proporcionarle como parámetro el objeto que se desea eliminar. El único requisito que debemos cumplir es que el objeto a eliminar tiene que tener una estructura `ObjectStateEntry` asociada. Piense que el funcionamiento de `DeleteObject` es similar al caso de la inserción: la llamada a `DeleteObject` marca la propiedad `State` de esta estructura con el valor `Deleted` para que, cuando el método `SaveChanges` sea invocado, el contexto de trabajo pueda determinar qué entidades se desean eliminar, y a partir de los valores que contenga esta estructura en sus propiedades `OriginalValues` y `CurrentValues` pueda generar una sentencia DELETE de SQL.

En el siguiente ejemplo, **code\ef4_ex_34**, podemos ver cómo realizar la eliminación de una entidad `Customer`, para ello nos basaremos en el mismo modelo simple utilizado hasta ahora.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    string firstName = "Lucia";
    Customer customer = ( from c in context.Customers
                          where
                        c.FirstName.ToLower() ==
                            firstName.ToLower()
                         select c).FirstOrDefault();

    if ( customer != null )
    {
        context.Customers.DeleteObject(customer);
        context.SaveChanges();
    }
}
```

Si revisáramos que ocurre en el `ObjectStateManager` asociado al contexto de trabajo, podríamos ver como el estado de la entidad consultada, customer, cambia de `Unchanged` a `Deleted` una vez invocado el método `DeleteObject`.

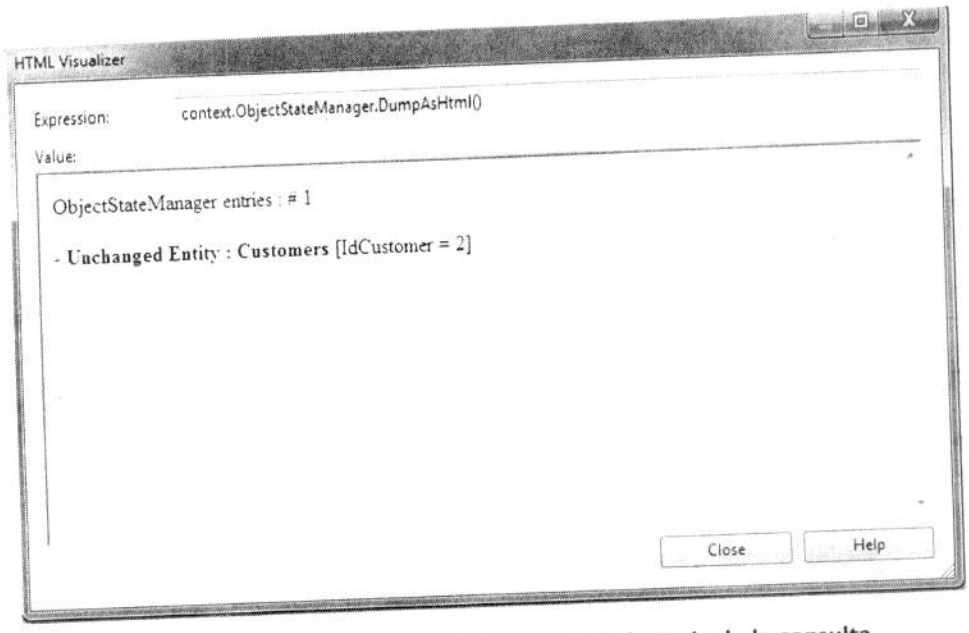


Figura 4.23.- ObjectStateEntry asociada después de la consulta

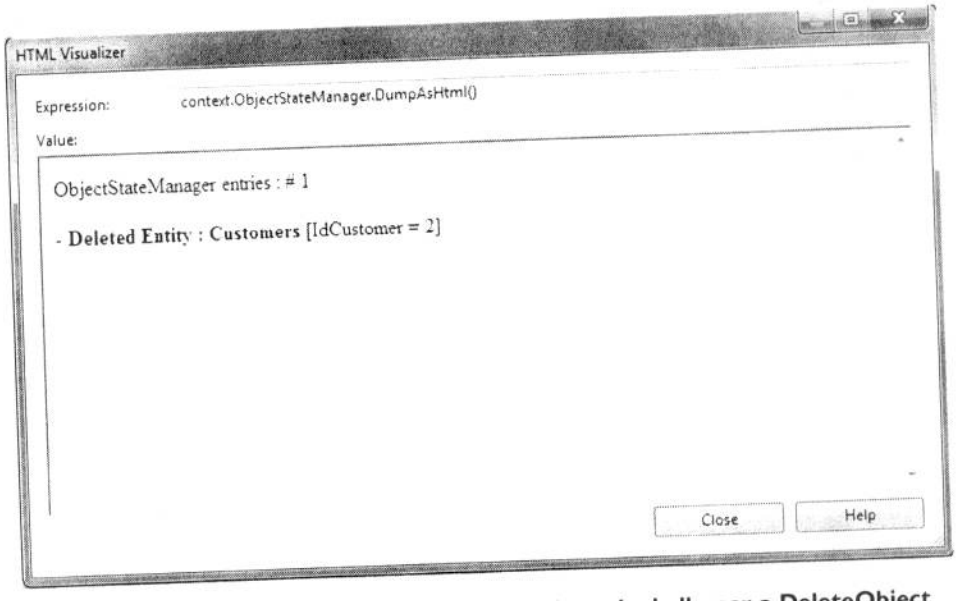


Figura 4.24.- ObjectStateEntry asociada después de llamar a DeleteObject

Nota: Por supuesto la llamada a `SaveChanges` podría fallar si se rompiera alguna integridad referencial dentro de la base de datos. En este caso, la infraestructura de ADO.NET Entity Framework nos notificaría el problema por medio de una excepción de tipo `UpdateException`.

Si modificáramos el ejemplo anterior sustituyendo la línea de consulta por la de una instancia generada por código de un objeto `Customer`, veríamos como, efectivamente, obtendríamos una excepción en la llamada a `DeleteObject` indicándonos que la entidad `Customer` que estamos intentando eliminar no tiene una estructura `ObjectStateEntry` asociada, y por tanto no se puede establecer su estado a `Deleted`.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    Customer customer = new Customer()
    {
        IdCustomer = 2,
        FirstName ="Lucia"
    };

    if (customer != null)
    {
         context.Customers.DeleteObject(customer);
         context.SaveChanges();
     }
}
```

Para resolver estas situaciones, en las que estamos intentando eliminar objetos que no han sido consultados previamente por el contexto, debemos informar a nuestro contexto de trabajo que comience a realizar seguimiento para esta entidad y por lo tanto cree una estructura de tipo `ObjectStateEntry` asociada a ella. Este trabajo se puede realizar fácilmente mediante una llamada al método `Attach` de nuestros `ObjectSet<TEntity>`.

```
public void Attach(TEntity entity);
```

A continuación puede ver la modificación del código anterior haciendo uso del método `Attach`. En la figura 4.25 se puede observar cómo después de la llamada a nuestro método `Attach` el `ObjectStateManager` asociado ya contiene una entrada de seguimiento para esta entidad, lógicamente en estado `Unchanged`.

```
using (EF4EXEntities context = new EF4EXEntities())
{
     Customer customer = new Customer()
     {
         IdCustomer = 2,
         FirstName ="Lucia"
     };
     context.Customers.Attach(customer);

     if (customer != null)
     {
          context.Customers.DeleteObject(customer);
          context.SaveChanges();
     }
}
```

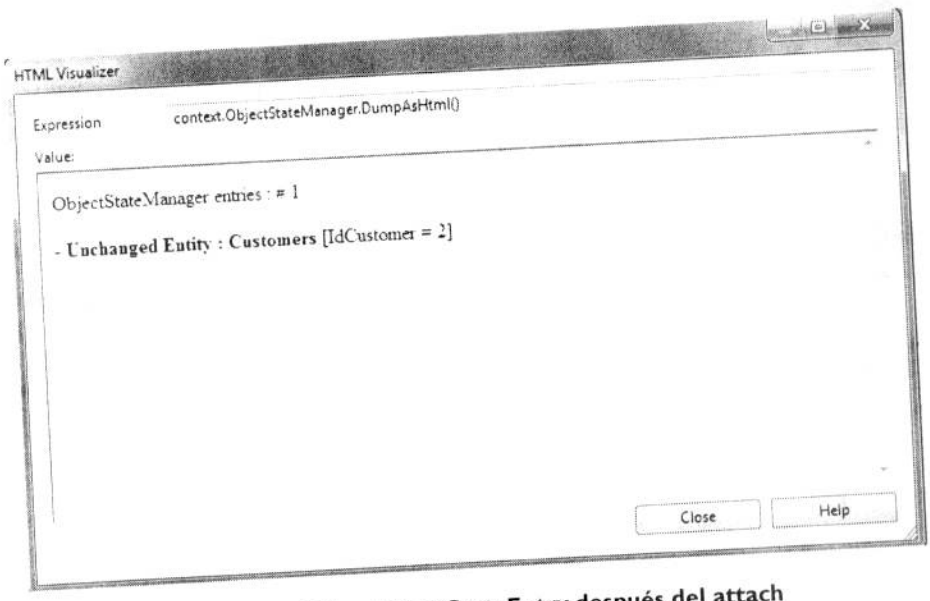


Figura 4.25.- ObjectStateEntry después del attach

Nota: Hasta ahora, en todos y cada uno de nuestros ejemplos, siempre hemos estado trabajando con un contexto de trabajo construido in-situ, creándolo dentro de una sentencia using, y por lo tanto usándolo y liberando sus recursos con la llamada implícita al método Dispose. Este patrón de uso de los contextos de trabajo, conocido como contexto de corto tiempo de vida o de corta duración (short lived contexts) implica la destrucción también del trabajo de seguimiento y consistencia del grafo realizado por ObjectStateManager,. Así, entender el porqué de la técnica anterior resulta esencial cuando intentamos eliminar objetos que no están presentes en un contexto, bien porque se han creado mediante el operador new o porque son elementos obtenidos de otro contexto de trabajo diferente. El uso de contextos de corta duración (short lived contexts) es algo habitual en las aplicaciones distribuidas; piense que el mantenimiento de contextos de larga duración (long running contexts) supone un abuso de recursos que no se puede asumir en la mayoría de las ocasiones.

Aun así, hay escenarios en los que el uso de un único contexto de trabajo tiene sentido, por ejemplo en clientes "pesados". El cómo trabajar correctamente con escenarios de contextos de corta duración será tratado más adelante, en este mismo capítulo, cuando mostremos la relación de la gestión del estado en entidades desconectadas y nuestras Selft Tracking Entities, presentadas anteriormente al comienzo de este capítulo.

Si se fija nuevamente en los métodos de nuestro tipo `ObjectSet<TEntity>`, podrá observar como además del mencionado método `Attach` también disponemos de un método `Detach`. como se imaginará esto método nos permitirá eliminar el seguimiento de una entidad por parte del `ObjectStateManager`. Dejaremos como ejercicio al lector comprobar esta característica.

```
public void Detach(TEntity entity);
```

En la sección anterior, relacionada con la inserción de entidades, vimos como cuando agregamos una entidad a un contexto podemos agregar además, con una sola llamada, todo el grafo de relaciones asociado. En las siguientes líneas veremos si este comportamiento es simétrico a la hora de la eliminación de entidades, o si por el contrario necesitamos realizar otro tipo de operaciones.

Para comprobarlo incluiremos valores en nuestra base de datos para nuestra entidad `Order`, y trataremos de realizar la eliminación del *cliente* asociado a esos pedidos. Ante este caso podrá observar como el sistema nos devuelve una excepción de tipo `UpdateException` conteniendo un mensaje descriptivo acerca de la rotura de la integridad referencia de las tablas `Customer` y `Order` de nuestra base de datos. Esto es lógico, ADO.NET EF no conoce nada acerca de los elementos de tipo `Order` relacionados con nuestro cliente y por lo tanto, no puede hacer nada con ellos, delegando la operación en la base de datos.

Por supuesto, el uso de una restricción **ON DELETE CASCADE** en nuestra base de datos podría ayudarnos mucho ya que nos libraría de la excepción obtenida. Sin embargo, también es posible que no tengamos esta posibilidad o que no podamos tocar el modelo relacional de la base de datos con la que estamos trabajando. Ante esta situación solamente nos quedará la opción de implementar por programa nuestro borrado en orden, como se observa en el siguiente fragmento:

```
using (EF4EXEntities context = new EF4EXEntities())
{
     Customer customer = (from c in context.Customers
                          select c).FirstOrDefault();

     foreach(Order order in customer.Orders.ToList())
     {
         foreach(OrderDetail detail in
                   order.OrderDetails.ToList())

            context.OrderDetails.DeleteObject(detail);

         context.Orders.DeleteObject(order);
     }


       context.Customers.DeleteObject(customer);

       context.SaveChanges();
}
```

Las figuras 4.26 y 4.27 nos permiten ver el estado del `ObjectStateManager` justamente antes de la llamada a `SaveChanges` y las sentencias ejecutadas vistas desde nuestra ventana de `Intellitrace`.

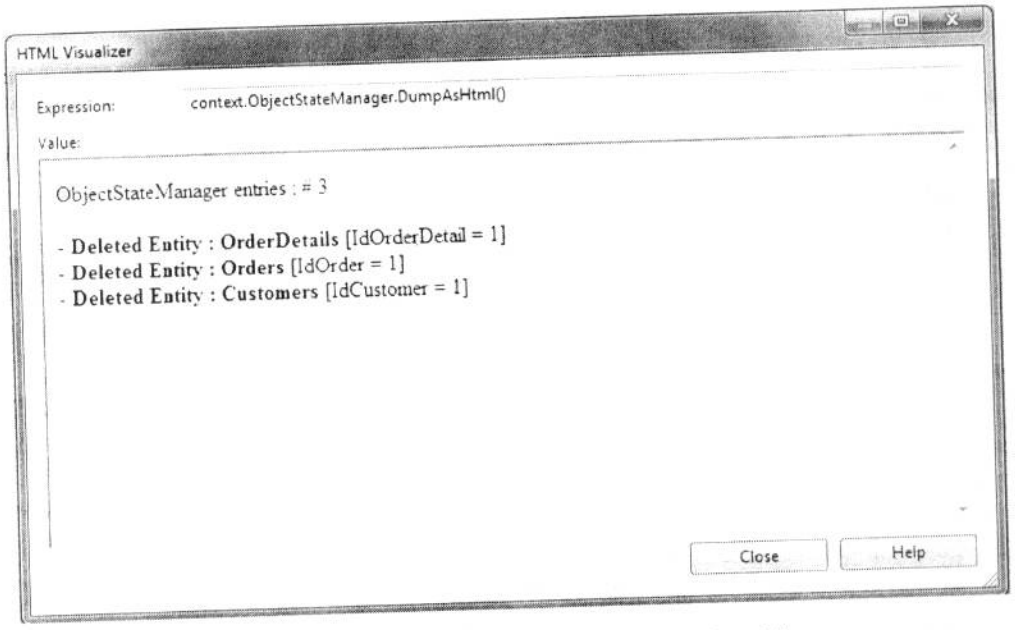


Figura 4.26.- ObjectStateManager antes de SaveChanges

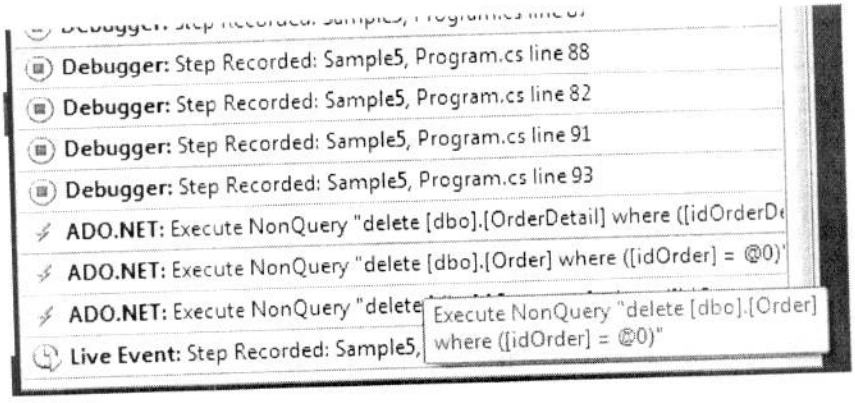


Figura 4.27.- Sentencias ejecutadas contra la base de datos

Nota: Implementar un borrado en cascada fuera de la base de datos podría llegar a lanzar multitud de consultas a la base de datos, imagínese borrar pedidos de un cliente. A menudo es usual recurrir a borrados lógicos, no solamente para evitar estos problemas sino también para evitar pérdida de consistencia de datos.

La eliminación de elementos es el primer caso en el que nos podríamos encontrar con problemas de concurrencia. Es posible que estemos intentando eliminar un elemento que ya no existe, o querramos asegurarnos de que el elemento a eliminar es el mismo que el que se encuentra en ese momento en la base de datos; o sea, que nadie lo ha modificado respecto a cuando lo leímos. Con la llegada de la primera versión de ADO.NET, este problema se puso de manifiesto a la hora de trabajar con modelos desconectados, y veíamos como los `DataSet` nos permitían de una forma muy sencilla resolverlo, modificando las sentencias SQL para incluir, dentro de la sentencia **`WHERE`**, las comparaciones de la fila actual en la base de datos con la fila en el objeto `DataTable` sobre el que realizábamos la operación. EF también nos permite implementar operaciones que tengan en cuenta el problema de la concurrencia, incluso con algunas mejoras adicionales que iremos comentando en las siguientes lineas.

Cada una de las propiedades de las entidades presentes en nuestros modelos conceptuales, además de las otras características que poseen que presentamos en el Capítulo 2, nos ofrecen la posibilidad de establecer cómo las mismas se tendrán en cuenta a la hora de manejar posibles problemas de concurrencia. `ConcurrencyMode`, una de las características que podemos establecer en las propiedades de una entidad, nos permitirá indicar si el valor de la propiedad debe tenerse en cuenta a la hora de implementar un mecanismo de concurrencia optimista; es decir, si esta propiedad deberá formar parte de la condición de la sentencia SQL de borrado. Fíjese en que el nivel de granularidad en EF es más fino que el del mecanismo utilizado en `DataSet`, puesto que con estos últimos no podíamos seleccionar de forma individual qué columnas formaban parte del **`WHERE`**.

En la siguiente figura, 4.28, puede ver como establecemos el valor de la propiedad `FirstName` con el atributo `ConcurrencyMode` a `Fixed`.

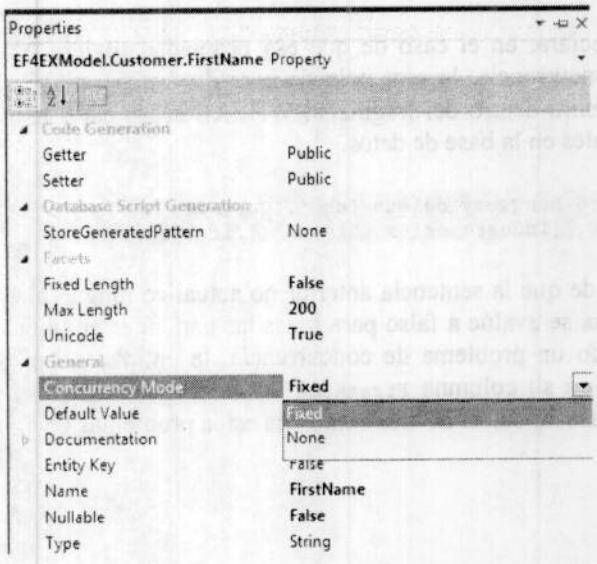


Figura 4.28.- Valor de ConcurrencyMode Fixed

La propiedad `ConcurrencyMode` puede establecerse a uno de dos valores, `Fixed` o `None` (el valor por defecto). El valor `None` nos permite indicar que el valor de la propiedad en cuestión no se tomará en cuenta para resolver un posible problema de concurrencia. Por el contrario, el valor `Fixed` indica que el valor de esta propiedad debe tenerse en cuenta a la hora de realizar operaciones en las que se pueda producir un posible problema de concurrencia, como un borrado o una actualización.

Si recuerda el trabajo que se hacía en los `DataSet` para resolver este problema, el asistente de generación de comandos agregaba cláusulas de comparación dentro de las expresiones **WHERE** para las sentencias de borrado y actualización. Estas sentencias eran similares a "`WHERE Prop=@Original_Prop`", donde esos valores originales eran obtenidos del propio `DataSet`, gracias a su `Diffgram`. EF utiliza esta misma técnica, aunque ahora es el contexto de trabajo el único que puede conocer los valores originales de una entidad, mediante el `ObjectStateEntry` asociado.

Como ejemplo práctico de lo que hemos comentado, **code\ef4_ex_35**, presentaremos un ejercicio cuyo objetivo es comparar las sentencias de eliminación de una entidad de tipo `Customer` con los valores de `ConcurrencyMode` establecidos como `Fixed` y `None` para su propiedad `FirstName`. El código se presenta a continuación:

```
using (EF4EXEntities context = new EF4EXEntities())
{
    Customer customer = context.Customers
                                .First();

    context.Customers
           .DeleteObject(customer);

    context.SaveChanges();
}
```

Para el caso en el que el valor de `ConcurrencyMode` está establecido a `None`, la sentencia de borrado se muestra a continuación.

```
Execute NonQuery delete [dbo].[Customer] where ([idCustomer] = @0
```

Como apreciará, en el caso de que esa propiedad tuviera establecido a `Fixed` el atributo de concurrencia, la sentencia de borrado nos permitirá comprobar mediante comparación extra dentro del fragmento WHERE si los datos actuales son coherentes con los originales en la base de datos.

```
Execute NonQuery delete [dbo].[Customer]
where (([idCustomer] = @0) and ([firstName] = @1
```

En el caso de que la sentencia anterior no actualice ningún elemento (o sea, que la expresión **WHERE** se evalúe a falso para todas las entidades en la base de datos), se dice que ha ocurrido un problema de concurrencia: la entidad o no existe en la base de datos, o tiene en su columna `FirstName` un valor diferente del que tenía cuando se obtuvo de la base de datos. EF nos comunica estos problemas de concurrencia lanzando

una `OptimisticConcurrencyException`, excepción que deberemos tratar para resolver la situación.

Hasta ahora, la forma tradicional de resolver un problema de concurrencia consistía en comunicárselo al usuario, con el fin de que éste refrescara los datos que estaba modificando y volviera a proceder a ejecutar la operación, o bien rechazar la operación, desde el propio mensaje informativo. EF pone a nuestra disposición una solución mucho más sencilla a través del método `Refresh` de `ObjectContext`, gracias al cual podremos obligar al motor de base de datos a aceptar nuestros cambios o bien hacer que los datos que contiene la base de datos sean considerados como los originales para el contexto de trabajo.

El método `Refresh` nos permitirá indicarle mediante parámetros la entidad o entidades que deseamos refrescar y un valor del tipo enumerado `RefreshMode`, mediante el que podremos especificar el modo de refresco, con el que decidimos si "ganan" los datos del cliente que realiza la operación o, por el contrario, los datos presentes en la base de datos.

```
public enum RefreshMode
{
    StoreWins = 1,
    ClientWins = 2,
}
```

El valor `ClientWins` de la enumeración indica que los datos actuales de la entidad que estamos intentando eliminar deben considerarse válidos y que, por lo tanto, el borrado debe efectuarse. Por el contrario, `StoreWins` indica que la operación no debe realizarse y que la entidad debe refrescar el valor de sus propiedades con los que hay actualmente en la base de datos.

Si nos detenemos un momento en ambas opciones, es muy probable que se dé cuenta de que esto concuerda en cierta manera con las opciones que teníamos disponibles en la enumeración `MergeOption`, vista en la sección dedicada a `ObjectStateManager`. El método `Refresh` con la opción `ClientWins` no es más que un proceso en el que se vuelve a realizar la consulta a la base de datos sobre la entidad o entidades pasadas como parámetro con la opción `MergeOption.PreserveChanges`; es decir, se vuelven a consultar las entidades, manteniendo los valores de la propiedad `CurrentValues` y refrescando los valores de la propiedad `OriginalValues` en los `ObjectStateEntry` asociados con los que hay en la base de datos.

Por el contrario, el valor `StoreWins` establece como valor para `MergeOption` la opción `OverwriteChanges`, con lo que los valores actuales, modificados o no, se cambian por los valores obtenidos en la consulta y el estado del `ObjectStateEntry` asociado se establece como `Unchanged`.

El siguiente fragmento de código muestra un ejemplo de uso del método `Refresh`, en este caso con la opción `ClientWins`.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    string firstName = "Maria";
    Customer customer = context.Customers
            .Where(c=>c.FirstName.ToLower()
```

```
                    ==
                firstName.ToLower())
        .First();

    try
    {
        context.Customers
               .DeleteObject(customer);

        context.SaveChanges();

    }
    catch (OptimisticConcurrencyException ex)
    {
            //call to refresh invalid OSE
            // (Object State Entry)

    context.Refresh(System.Data.Objects.RefreshMode.ClientWins,
                ex.StateEntries.Select(e => e.Entity));

                context.SaveChanges();
        }
    }
}
```

Si hay un problema de concurrencia cuando se llama al método `SaveChanges`, se produce una excepción de tipo `OptimisticConcurrencyException`. El código de tratamiento hace una llamada al método `Refresh`, el cual provoca que se envíe una sentencia de selección a la base de datos en busca de los datos que ésta contiene para el `Customer` que estamos tratando de eliminar. Los datos obtenidos se establecen como datos originales del distribuidor (algo esencial si no queremos que la consulta de eliminación vuelva a fallar), conservando los valores de los datos actuales. Puesto que la estructura `ObjectStateEntry` correspondiente sigue con el valor `Modified`, es necesario volver a realizar una llamada al método `SaveChanges`, ahora sin miedo ninguno a que esta sentencia **DELETE** falle por un problema de concurrencia.

2.5.4.- Actualización de entidades

La actualización de entidades es la última de las operaciones DML tradicionales de un motor de base de datos que debemos tratar aquí. Al igual que en los casos anteriores, la actualización se realiza de una forma fuertemente tipada, y siempre manejamos clases y propiedades de las mismas. Empezaremos de nuevo por un ejemplo sencillo, y posteriormente iremos comentando las distintas problemáticas y soluciones con respecto a la actualización de entidades.

En el primer ejemplo, que se muestra a continuación, vemos cómo obtener un elemento de tipo `Customer` y realizar una modificación (con la subsiguiente actualización en el almacén) de su propiedad `FirstName`.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    string firstName = "Maria";
```

```
        Customer customer = (from c in context.Customers
                             where c.FirstName.ToLower()
                                   ==
                                   firstName.ToLower()
                             select c).First();

        customer.FirstName = "Modificación del valor!";

        context.SaveChanges();
}
```

Como puede apreciar, este proceso es totalmente natural para un programador orientado a objetos: siempre se trabaja de una forma tipada, y actualizar una entidad es algo tan simple como asignar un valor a la propiedad que queremos modificar.

Internamente, el proceso es idéntico al de los casos anteriores: cuando una propiedad de una entidad es modificada, el estado del `ObjectStateEntry` asociado a la misma se establece de forma automática a `Modified`. La posterior llamada a `SaveChanges` hace que el contexto, por medio del `ObjectStateManager`, envíe al almacén relacional una sentencia UPDATE de SQL en función de los valores actuales y originales de la estructura.

Como usted sabe la actualización de entidades es el segundo de los casos en los que podríamos tener problemas de concurrencia. Estos posibles problemas se resuelven de igual forma que durante la eliminación de entidades, mediante el uso del método `Refresh`; un ejemplo se presenta a continuación:

```
using (EF4EXEntities context = new EF4EXEntities())
{
    string firstName = "Maria";
    Customer customer = (from c in context.Customers
                         where c.FirstName.ToLower()
                               ==
                               firstName.ToLower()
                         select c).First();

    customer.FirstName = "Modificación del valor!";

    try
    {
        context.SaveChanges();
    }
    catch (OptimisticConcurrencyException ex)
    {
        context.Refresh(RefreshMode.ClientWins,
                        ex.StateEntries.Select(e
                                               =>
                                               e.Entity));

        context.SaveChanges();
    }
}
```

El caso de la actualización de entidades es el más complicado de tratar, dependiendo del tipo de uso de los contextos de trabajo. En principio, si utilizamos un

contexto de larga duración (uno por aplicación, por poner un ejemplo claro), nunca tendríamos problemas, pero como hemos visto por ejemplo para la eliminación de entidades, en los contextos de vida cortos necesitamos que el contexto de trabajo tenga una estructura `ObjectStateEntry` asociada a los objetos que estamos manejando, razón por la cual recurríamos al método `Attach`. Este método no es suficiente para resolver el problema de actualizar un elemento que proviene de otro contexto de trabajo, o que ha viajado, serializado, desde un cliente hasta una fachada de servicios. El motivo es que, al tratar únicamente con los objetos, en ningún momento el contexto puede averiguar qué propiedades del objeto se han modificado y cuáles no y, por tanto, no puede saber qué sentencia de actualización tendría que ejecutar contra la base de datos subyacente.

Tal y como sabemos, el estado de los objetos, así como sus valores originales, no es algo inherente a los objetos; estos datos son parte de las estructuras `ObjectStateEntry` almacenadas en el contexto de trabajo. Por lo tanto, para resolver la problemática anterior deberíamos ser capaces, en primer lugar, de crear una estructura `ObjectStateEntry` para un objeto no consultado por el contexto, y posteriormente ser capaces de indicarle al contexto qué propiedades han sido modificadas y qué valores se han establecido. Para implementar estos mecanismos, dentro del tipo `ObjectStateManager` tenemos distintos métodos que nos permitirían implementar algún mecanismo para resolver estos problemas.

```
public TEntity ApplyCurrentValues<TEntity>(string entitySetName,
TEntity currentEntity) where TEntity : class;

public TEntity ApplyOriginalValues<TEntity>(string entitySetName,
TEntity originalEntity) where TEntity : class;

public void ApplyPropertyChanges(string entitySetName, object
changed);
```

Realmente, aunque podríamos avanzar aquí en intentar mostrar un ejemplo para manejar los problemas de actualizaciones en **contextos de vida corto,** tal y como hicimos en la versión anterior del libro, creemos que ya no es necesario debido a la inclusión dentro de **ADO.NET Entity Framework** de nuestras entidades de seguimiento propio o **Selft Tracking Entities**, sobre las cuales hablaremos más adelante.

2.5.5.- Transaccionabilidad en los servicios de objetos

Todos los cambios que se realizan sobre un contexto de objetos son aplicados de manera transaccional, de modo que los cambios registrados en el contexto de datos se aplicarán en la base de datos subyacente bajo el principio "o todos o ninguno". Si se da cuenta de que no hay ninguna transacción activa en ese momento, EF crea una transacción y encapsula la llamada a `SaveChanges` dentro de ella. Por otra parte, si en el momento de la aplicación de los cambios ya existe una transacción activa, EF utilizará esa transacción, hecho que puede ser utilizado por el programador en caso de que el comportamiento predeterminado no le satisfaga. Por ejemplo, si tuviéramos a nuestra

disposición dos contextos de objetos `ctx1` y `ctx2` y quisiéramos implementar una transacción distribuida sobre las dos bases de datos subyacentes a esos contextos, podríamos utilizar las clases del ensamblado `System.Transactions` de la siguiente forma:

```
using (TransactionScope ts = new TransactionScope())
{
    ctx1.SaveChanges();
    ctx2.SaveChanges();


    ts.Complete();
}
```

La otra posible vía de manejar las transacciones desde los servicios de objetos sería accediendo al objeto `EntityConnection` asociado al contexto de objetos mediante la propiedad `Connection` de éste, y a través de la conexión gestionar las transacciones de manera similar a como lo hemos hecho en el capítulo correspondiente a `Entity Client`.

2.6.- Selft Tracking Entities y la gestión del estado

A lo largo del punto anterior hemos visto la importancia del tipo `ObjectStateManager` en ADO.NET Entity Framework y como el trabajo realizado por esta pieza es fundamental para que las distintas operaciones **DML** realmente funcionen correctamente. En varios de los puntos anteriores, se mencionó la dificultad de trabajo en escenarios de contextos de vida corta, **short lived context**, puesto que una vez que las entidades no están ligadas a un contexto cualquier operación sobre las mismas (como podría ser una modificación de una propiedad o cualquier variación en alguna de sus navegaciones: agregar, modificar o eliminar cualquier de los elementos asociados), no es seguida por ningún elemento, y, por lo tanto, no sabríamos como replicarla en una nueva instancia de un contexto de trabajo en una posterior petición.

Para resolver esta problemática, normal en cualquier aplicación distribuída que intentemos hacer con **ADO.NET EF**, en esta nueva versión del producto, disponemos de nuestras entidades **Selft Tracking Entities** o entidades de seguimiento propio.

Ya hablamos un poco sobre este tipo de entidades al comienzo de este capítulo, en el que enseñamos la forma de las mismas, haciendo una pequeña comparación con las clases prescriptivas que EF genera por defecto. En esta descripción de las entidades STE, se presentó la interfaz `IObjectWithChangeTracker`, aunque retardamos la explicación de su funcionalidad hasta este instante. La principal capacidad de estas entidades, proporcionada por esta interfaz, es que las mismas pueden realizar el seguimiento de las acciones que sobre ellas se realizan, como una modificación de alguna de sus propiedades o bien la modificación (inserción, borrado o actualización) de alguna de sus propiedades de navegación.

Durante las siguientes líneas, trataremos en primer lugar de mostrar como se han construido estas entidades así como la forma de utilizarlas dentro de nuestras aplicaciones.

Una vez asignado **STE** como elemento de generación de entidades, tal y como se ha mostrado anteriormente, podremos observar como todas las entidades generadas se basan en la interfaz `IObjectWithChangeTracker`, presentada a continuación:

```
public interface IObjectWithChangeTracker
{
    ObjectChangeTracker ChangeTracker { get; }
}
```

Esta interfaz, que se incorpora a nuestro código por medio de las plantillas **T4**, por lo tanto "independiente de las librerías de EF", define una propiedad `ChangeTracker` que nos permite obtener el tipo fundamental que realizará el seguimiento de la entidad en la que esté definida. Este tipo, cuya estructura básica podemos ver a continuación en la siguiente figura, **Figura 4.29**, define una serie de contenedores y métodos de ayuda para registrar distintas operaciones sobre la entidad y las distinas asociaciones de la misma.

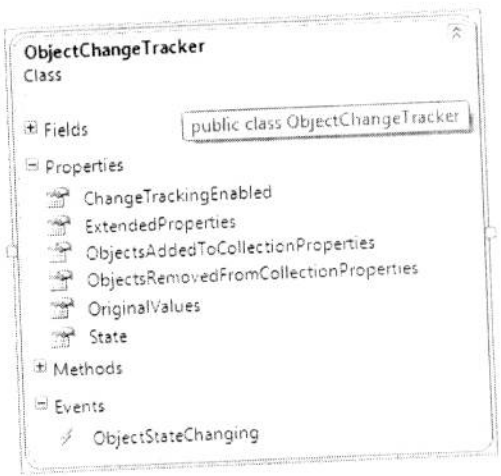


Figura 4.29.- Estructura del tipo ObjectChangeTracker

Para el tratamiento de las asociaciones disponemos de dos colecciones espcializadas, `ObjectsAddedToCollectionProrperties` y `ObjectsRemovedFromCollectionProperties`, cuya firma tenemos a continuación.

```
public class ObjectsAddedToCollectionProperties :
    Dictionary<string, ObjectList> { }

public class ObjectsRemovedFromCollectionProperties :
    Dictionary<string, ObjectList> { }
```

Si recuerda, al comienzo de este capítulo, en el punto 2.2.1, hablamos sobre el tipo `TrackableCollection` y como el mismo, gracias a su relación con `ObservableCollection`, permitía subscribirnos a sus cambios por medio de un evento.

Pues bien, cuando una propiedad de navegación es modificada, en el manejador del evento de cambio de este tipo de colleciones se hace uso de alguno de los dos tipos anteriores para almacenar el nuevo elemento agregado, o bien el elemento eliminado de la colección.

Para ilustrarlo mejor, en nuestro ejemplo, **code\ef4_ex_37**, podemos ver el manejador del evento de cambio de la propiedad `OrderDetails` de nuestra ya conocida entidad `Orders`:

```
private          void          FixupOrderDetails(object          sender,
NotifyCollectionChangedEventArgs e)
{
    if (IsDeserializing)
    {
        return;
    }
    if (e.NewItems != null)
    {
       foreach (OrderDetail item in e.NewItems)
       {
           item.Order = this;
           if (ChangeTracker.ChangeTrackingEnabled)
           {
              if (!item.ChangeTracker.ChangeTrackingEnabled)
              {
                 item.StartTracking();
              }

ChangeTracker.RecordAdditionToCollectionProperties("OrderDetails",
item);
           }
       }
    }

    if (e.OldItems != null)
    {
    foreach (OrderDetail item in e.OldItems)
    {
       if (ReferenceEquals(item.Order, this))
       {
          item.Order = null;
       }
       if (ChangeTracker.ChangeTrackingEnabled)
       {

ChangeTracker.RecordRemovalFromCollectionProperties("OrderDetails",
item);
       }
    }
  }
}
```

Como observará, aquellos elementos introducidos en nuestra propiedad OrderDetails, definidos en este manejador por medio de la propiedad `e.NewItems`, son incluidos en la colección de elementos agregados por medio de la llamada al método de ayuda `RecordAdditionToCollectionProperties`. Del mismo modo, aquellos elementos

eliminados, obtenidos por medio de `e.OldITems` son agregados a la colección de elementos eliminados por medio del método `RecordRemovalFromCollection-Properties`.

Por supuesto, además de registrar cambios en las propiedades de navegación, nuestras entidades STE también nos permiten registrar la modificación de las distintas propiedades primitivas de la misma. En el siguiente fragmento, puede ver el código de la propiedad `OrderDate`.

```
[DataMember]
public System.DateTime OrderDate
{
    get { return _orderDate; }
    set
    {
        if (_orderDate != value)
        {
            _orderDate = value;
            OnPropertyChanged("OrderDate");
        }
    }
}
```

En esta propiedad, cuando se establece un valor, por medio de su método `Set`, se notifica el cambio usando para ello el método `OnPropertyChanged`, mostrado a continuación. En este método, como aprecia, se comprueba el estado de la entidad, asegurándose que no se está estableciendo el valor para una propiedad nueva o marcada para eliminar.

```
protected virtual void OnPropertyChanged(String propertyName)
{
    if (ChangeTracker.State != ObjectState.Added &&
ChangeTracker.State != ObjectState.Deleted)
    {
        ChangeTracker.State = ObjectState.Modified;
    }
    if (_propertyChanged != null)
    {
        _propertyChanged(this, new
PropertyChangedEventArgs(propertyName));
    }
}
```

Los distintos estados de una entidad, como podemos ver en el código anterior, están marcados por el enumerado `ObjectState`. En este enumerado se definen los estados siguientes:

```
[Flags]
public enum ObjectState
{
    Unchanged = 0x1,
    Added = 0x2,
    Modified = 0x4,
    Deleted = 0x8
}
```

Por defecto, nuestra clase `ObjectChangeTracker` establece el valor del estado como *Added*. Sin embargo, cuando una entidad es materializada por medio de un contexto de trabajo de ADO.NET Entity Framework, se realiza un trabajo adicional -como se puede ver a continuación- que consiste en establecer este valor de estado a `Unchanged`. Adicionalmente también se establece una bandera que nos permite habilitar/deshabilitar el seguimiento de las entidades.

```
private void HandleObjectMaterialized(object sender,
ObjectMaterializedEventArgs e)
{
    var entity = e.Entity as IObjectWithChangeTracker;
    if (entity != null)
    {
       bool changeTrackingEnabled =
entity.ChangeTracker.ChangeTrackingEnabled;
       try
       {
           entity.MarkAsUnchanged();
       }
       finally
       {
                   entity.ChangeTracker.ChangeTrackingEnabled =
changeTrackingEnabled;
       }
       this.StoreReferenceKeyValues(entity);
    }
}
```

La llamada a `MarkAsUnchanged` establece el valor de la propiedad `ObjectState` como `UnChanged`. Al igual que este método extensor, dentro de nuestra plantilla STE se definen otros métodos extensores para modificar el estado de una entidad rápidamente, `MarkAsAdded` o `MarkAsDelete` son otros ejemplos de los métodos extensores existentes.

Llegados hasta aquí, de las entidades `STE` hemos visto como almacenan los cambios de sus propiedades de navegación por medio de unas nuevas colecciones y nuestras propiedades `TrackableCollection`. A mayores, también hemos visto el uso del enumerado `ObjectState` y como cuando una entidad se materializa dicho valor se establece como `UnChanged`. El *flag* `ChangeTrackingEnabled`, como hemos comentado, nos permite establecer u obtener un valor indicándonos si la entidad esta "realizando el seguimiento". Fíjese como el código anterior de manejo de cambios en propiedades de navegación, `FixupOrderDetails`, realiza una comprobación de este flag.

Por defecto, cuando una entidad es recuperada y materializada por medio de un contexto de trabajo, el valor de esta bandera está establecido como falso, por lo que no se realizará seguimiento. Aunque es posible, por supuesto, iniciar el seguimiento estableciendo a *true* el valor de `ChangeTrackingEnabled` o bien usando el método extensor `StartTracking`, presentado a continuación, realmente la activación del seguimiento se realizará cuando las entidades sean serializadas.

```
public static void StartTracking(this IObjectWithChangeTracker
trackingItem)
{
    if (trackingItem == null)
```

```
    {
        throw new ArgumentNullException("trackingItem");
    }

    trackingItem.ChangeTracker.ChangeTrackingEnabled = true;
}
```

Nota: Aunque el código de las propiedades primitivas de una entidad en su método `set` establezcan el valor del estado a `Modified`, el tipo `ChangeTracker` exige que el seguimiento de cambios esté activado.

```
[DataMember]
public ObjectState State
{
    get { return _objectState; }
    set
    {
        if (_isDeserializing || _changeTrackingEnabled)
        {
            OnObjectStateChanging(value);
            _objectState = value;
        }
    }
}
```

Aunque esto le pueda sonar extraño, en realidad tiene todo el sentido del mundo puesto que estas entidades están pensadas para trabajar en entornos distribuidos. De hecho, mucho del código asociado a estas entidades es robusto en cuanto a la serialización, es decir, que el proceso de serialización y deserialización no modifican ni alteran el estado de las entidades y/o de sus navegaciones.

Para comprobar como se modifica el seguimiento de una entidad gracias al proceso de serialización, realizaremos un sencillo ejemplo, para ello utilizaremos el siguiente método que nos permitirá serializar/deserializar cualquier entidad STE.

```
static TEntity Serialize<TEntity>(TEntity item) where TEntity:class
{
    DataContractSerializer serializer = new
DataContractSerializer(typeof(TEntity));
    using (MemoryStream stream = new MemoryStream())
    {
        serializer.WriteObject(stream, item);
        stream.Seek(0, SeekOrigin.Begin);

        return serializer.ReadObject(stream) as TEntity;
    }
}
```

En nuestro ejemplo, a continuación, mostraremos los valores de la bandera `ChangeTrackingEnable` y del estado de una entidad antes y después de un proceso de serialización.

```
using (EF4EXEntities context = new EF4EXEntities())
{
     Order order = context.Orders.First();

     Console.WriteLine("TrackingEnabled:{0}
State:{1}",order.ChangeTracker.ChangeTrackingEnabled,

order.ChangeTracker.State);

      Order serializedOrder = Serialize(order);
      Console.WriteLine("TrackingEnabled:{0} State:{1}",

serializedOrder.ChangeTracker.ChangeTrackingEnabled ,

serializedOrder.ChangeTracker.State);

}
```

Una vez ejecutado este código, los resultados obtenidos por pantalla serían los siguientes:

```
TrackingEnabled:False State:Unchanged
TrackingEnabled:True State:Unchanged
```

Llegados hasta aquí, hemos visto internamente como están construídas nuestras entidades STE, y como los distintos elementos incorporados junto a ellas nos permiten realizar el seguiemiento de las entidades.

Lógicamente, una vez que disponemos de este seguimiento, deberíamos disponer de algún mecanismo que nos permitiera adaptar esta información a nuestro `ObjectStateManager`. Es decir, una vez que las entidades han viajado a nuestro cliente, se han modificado (por modificación entendemos cualquier proceso como la adición, substracción o alteración de algunos de los valores de sus propiedades) y han vuelto de nuevo, cómo podemos incluir toda la información de seguimiento dentro de nuestros contextos de trabajo. Pues bien, para este trabajo, la plantilla de STE, además del propio código de los contextos de trabajo incluye una serie de métodos extensores de ayuda. En concreto, el que nos interesa a nosotros es un método extensor sobre el tipo `ObjectSet<TEntity>`, que seguro ya le sonará familiar, llamado `ApplyChanges`, cuya firma puede verse a continuación.

```
public static void ApplyChanges<TEntity>(this ObjectSet<TEntity>
objectSet, TEntity entity) where TEntity : class,
IObjectWithChangeTracker
```

Hacer uso de este método es realmente sencillo, basta con invocarlo desde el `ObjectSet` asociado al elemento que contiene las modificaciones. A continuación, también dentro del ejemplo, **code\ef4_ex_37**, podemos ver una pequeña muestra de todo el proceso.

```
Order order, clientOrder, serverOrder;

//Obtenemos un elemento de tipo order
using (EF4EXEntities context = new EF4EXEntities())
{
    order = context.Orders.First();
}
//lo serializamos y deserializamos
//equivalente al retorno de una
//llamada a un servicio de wcf
clientOrder = Serialize(order);

//modificamos
clientOrder.OrderDate = DateTime.UtcNow;

//lo serializamos y deserializamos
//equivalente a un envio como parámetro
//de un método de servicio wcf
serverOrder = Serialize(clientOrder);

using (EF4EXEntities context = new EF4EXEntities())
{
    context.Orders.ApplyChanges(serverOrder);
    context.SaveChanges();
}
```

Si observáramos con nuestro visor del `ObjectStateManager` el contenido del mismo antes (figura 4.30) y después (figura 4.31) de la llamada al método extensor `ApplyChanges`, podríamos observar como la información de modificación, el cambio en el valor de la propiedad `OrderDate` se reproduce correctamente en una nueva `ObjectStateEntry`, y por lo tanto la llamada a `SaveChanges` podrá realizar las operaciones necesarias.

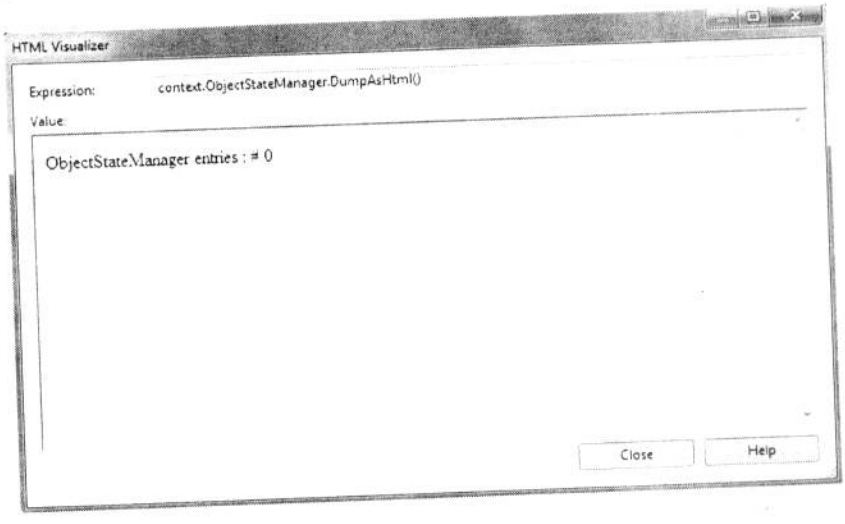


Figura 4.30.- Antes de la llamada a ApplyChanges

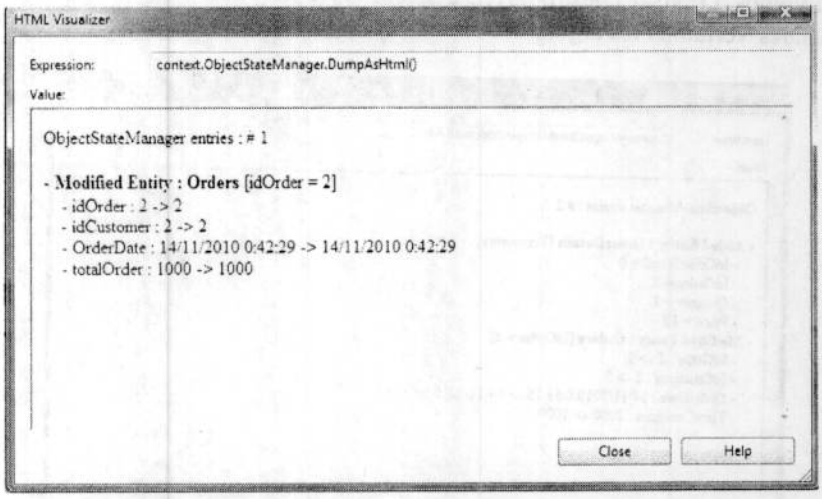


Figura 4.31.- Después de la llamada a ApplyChanges

Realmente, como obsevará en la figura 4.31, cuando una entidad llega en estado modificado todas las propiedades de la misma se marcan como modificadas, aunque alguna realmente no se hubiera tocado, como pasa en nuestro ejemplo.

Podríamos complicar más nuestras modificaciones en la parte cliente. Por ejemplo a nuestro elemento de tipo `Order` cambiarle quien es el cliente asociado así como agregar un detalle del pedido:

```
Order order, clientOrder, serverOrder;

//Obtenemos un elemento de tipo order
using (EF4EXEntities context = new EF4EXEntities())
{
    order = context.Orders.First();
}
//lo serializamos y deserializamos
//equivalente al retorno de una
//llamada a un servicio de wcf
clientOrder = Serialize(order);
clientOrder.IdCustomer = 3;
clientOrder.OrderDetails.Add(new OrderDetail()
                             {
                                 Price  =10,
                                 Quantity = 1
                             });

//modificamos
clientOrder.OrderDate = DateTime.UtcNow;

//lo serializamos y deserializamos
//equivalente a un envio como parámetro
//de un método de servicio wcf
serverOrder = Serialize(clientOrder);

using (EF4EXEntities context = new EF4EXEntities())
{
```

```
        context.Orders.ApplyChanges(serverOrder);
        context.SaveChanges();
    }
```

Si volvemos a revisar nuestro `ObjectStateManager` después de la llamada al método `ApplyChanges` veríamos los siguientes elementos, figura 4.32.

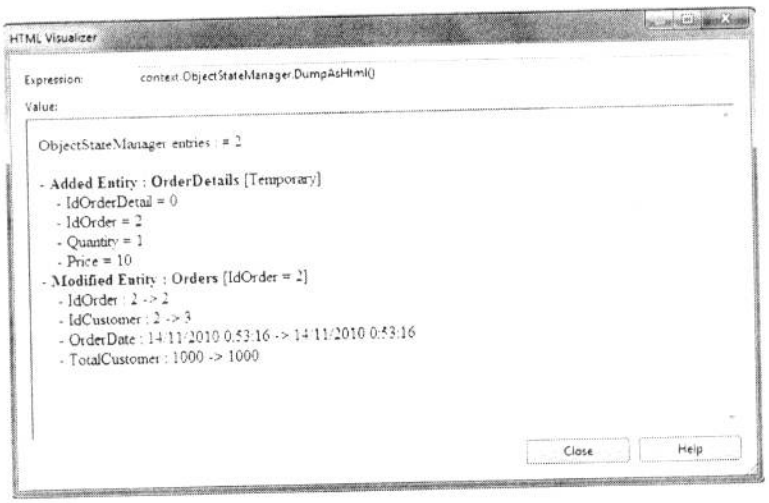


Figura 4.32.- Después de la llamada a ApplyChanges

Como habrá podido observar, el trabajo con las entidades **STE** es realmente sencillo, transparente para el desarrollador, puesto que no nos tenemos que preocupar de la gestión de los cambios en el cliente. Y después, cuando queramos guardar los mismos, solamente tendremos que hacer uso del método extensor `ApplyChanges`.

Puesto que todo el trabajo de "fontanería" forma parte de nuestras entidades, lógicamente, hay una serie de condicionantes que debemos de tener claros antes de seleccionar a STE como nuestras entidades del dominio.

En primer lugar como este código de `ChangeTracker` necesita viajar al cliente, en una arquitectura distribuída estamos implícitamente poniendo una restricción de tecnología, puesto que solamente podremos crear aplicaciones .NET a .NET.

Para terminar, aunque se podría argumentar que estas entidades mantienen el principio de ignorancia de la persistencia, (el código de `ChangeTracker` no depende de ADO.NET EF), realmente todo el trabajo de `ChangeTracker` está cargado de "conocimiento implícito" sobre ADO.NET EF.

2.6.1.- Entidades STE en Silverlight

Como se acaba de comentar, uno de los requisitos de uso de las entidades STE es que solamente pueden ser usadas desde .NET a .NET. A continuación, veremos el trabajo necesario para que las mismas estén disponibles en **Silverlight 4.1.**

Como usted sabrá, Silverlight es un subconjunto de .NET Framework, y por lo tanto, algunos de los elementos que usted utiliza en .NET Framework o bien no existen en Silverlight o su implementación es diferente. Por ello, la utilización de nuestras entidades STE dentro de Silverlight requiere de un trabajo extra. Aunque en principio se podría pensar, que gracias a la nueva capacidad de Silverlight 4.1, conocida como "assembly portability", el código de las STE podría ser establecido en un proyecto y ser usado tanto por Silverlight como por .NET Full Framework, por desgracia, algunos de los elementos como `DataContract` u `ObservableCollection<TEntity>` no son compatibles con esta nueva característica.

Nota: SL4 y .NET tiene los siguientes ensamblados marcados como portables:

- System
- System.Core
- System.ComponentModel.Composition
- Microsoft.VisualBasic

Si desea saber más acerca de la feature "asseembly portability" y de su utilización en Visual Studio 2010, le recomendamos el siguiente enlace:

http://blogs.msdn.com/b/clrteam/archive/2009/12/01/sharing-silverlight-assemblies-with-net-apps.aspx

Para salvar la limitación de uso del código de nuestras entidades STE, tanto en Silverlight como en .NET Framework, veremos como crear un proyecto de biblioteca de clases de Silverlight e incluir, también en el, la plantilla de generación de entidades.

En nuestro ejemplo, **code\ef4_ex_38**, partiremos de un modelo simple y el artefacto de generación de entidades STE. Posteriormente, agregaremos a nuesta solución un nuevo proyecto de librería de clases para Silverlight 4.1, figura 4.33.

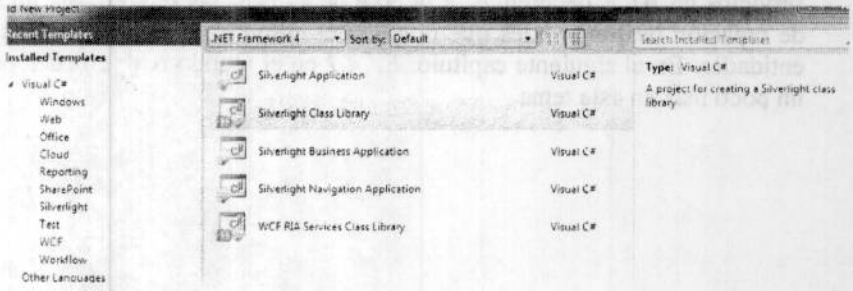


Figura 4. 33.- Creación del proyecto de librería de clases para Silverlight 4.1

Una vez que ya disponemos de nuestro proyecto, figura 4.34, agregaremos el elemento existente de nuestra plantilla de entidades, para ello podemos seleccionar desde el menú contextual de nuestro proyecto Silverlight la opción *"Agregar elemento existente (como Link)"*, figura 4.35.

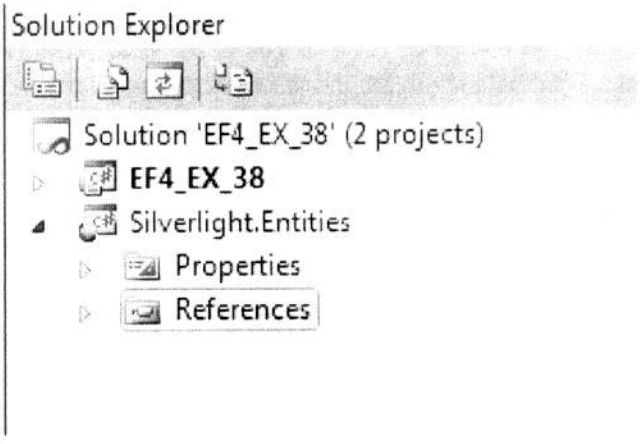


Figura 4.34.- Proyecto de SL en la solución

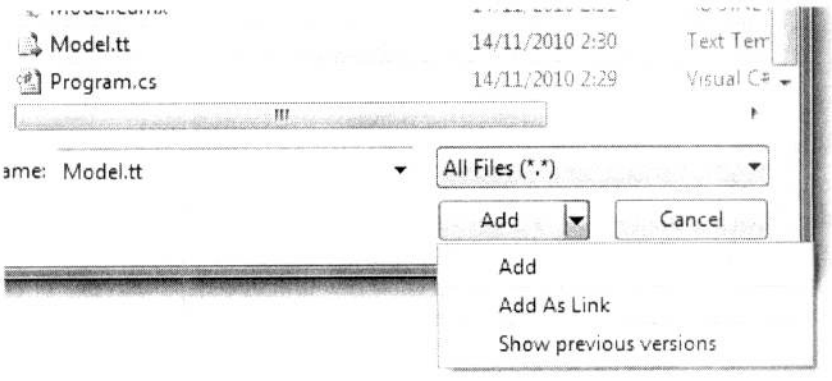


Figura 4.35.- Agregando la plantilla de STE como enlace en el proyecto de SL

Al agregar esta plantilla, el código generado por la misma (el mismo que en nuestro proyecto de .NET completo), se compilará correctamente en SL4 y por lo tanto ya lo podremos usar en nuestras aplicaciones. Por supuesto, si estamos trabajando con servicios de *WCF* deberemos, a la hora de agregar las referencias, indicar al asistente de creación de los proxies que reutilize el ensamblado creado en vez de generar las entidades. En el siguiente capítulo, *EF 4.1* en el mundo real, nos detendremos si cabe, un poco más en este tema

Nota: Una vez agregada la plantilla al proyecto de SL4 también es necesario agregar una referencia a System.Runtime.Serialization

Nota: Otro caso de uso de nuestras entidades STE podría ser en aplicaciones ASP.NET, en las mismas, podría ser necesario marcar nuestras entidades como *Binary Serializables*, algo que por defecto no son. Para realizar estos pasos, puesto que es muy mecánico y realmente no aporta nada al libro, le recomendamos seguir la siguiente entrada de blog del equipo de ADO.NET, dónde se detalla paso a paso lo que tendremos que realizar:

http://blogs.msdn.com/b/adonet/archive/2010/05/26/using-binary-serialization-and-viewstate-with-self-tracking-entities.aspx

De la misma forma que para incluir entidades dentro del *ViewState* de nuestras páginas ASP.NET tendremos que tocar nuestras plantillas T4.
Si queremos hacer uso de estas entidades dentro de *Windows Server AppFabric* tendremos también que realizar una pequeña modificación de nuestras plantillas. Desde aquí le recomendamos una entrada de blog de nuestro coautor Cesar de la Torre:

http://blogs.msdn.com/b/cesardelatorre/archive/2010/04/30/using-entity-framework-self-tracking-entities-with-appfabric-cache-velocity-and-net-4-0-rtm.aspx

2.7.- Entidades POCO y la gestión del estado

En el punto anterior hemos visto como el uso de nuestras entidades STE nos abre una enorme vía para trabajar en arquitecturas distribuídas con ADO.NET Entity Framework, dadas las increíbles aportaciones en productividad que las mismas incorporarn. Lógicamente, establecer un requisito de aplicaciones .NET a .NET no siempre es admisible dentro de la arquitectura de una aplicación, por lo tanto en no pocas ocasiones el uso de este tipo de entidades queda descartado por requisitos de interoperabilidad. Desde el grupo de producto, pensando en este tipo de casuísticas, entre otras, nos proporcionan la posibilidad de usar clases POCO (*Plain Old CLR Objects*) dentro de nuestros modelos de entidades con ADO.NET EF 4.1.

Anteriormente, ya hicimos una pequeña introducción acerca de la inclusión de las plantillas de generación, a continuación, veremos la relación que este tipo de entidades tienen con nuestro `ObjectStateManager` y como se realiza la gestión de cambios para este tipo de entidades.

Para nuestros ejemplos, utilizaremos el mismo modelo de entidades sencillo que en los puntos anteriores, **code\ef4_ex_39**. Lo primero que veremos es que información contiene nuestro `ObjectStateManager` de las entidades consultadas, si es que dispone de alguna. Para ello realizaremos la siguiente consulta y observaremos, con el visualizador HTML presentado anteriormente, la información de seguimiento:

```
using (EF4EXEntities context = new EF4EXEntities())
{
    Customer customer = context.Customers.FirstOrDefault();
}
```

Despues de esta consulta, lógicamente antes de salir del ámbito del `using` de nuestro contexto, si observarmos el contenido de nuestro OSM podremos ver, figura 4.36, como el contexto de trabajo realiza seguimiento de nuestras entidades. La verdad, es que esto no es muy difícil de entender, puesto que cuando realizamos una consulta, nuestro contexto de trabajo puede hacer, como hasta ahora, una nueva entrada `ObjectStateEntry` asociada a cada entidad materializada.

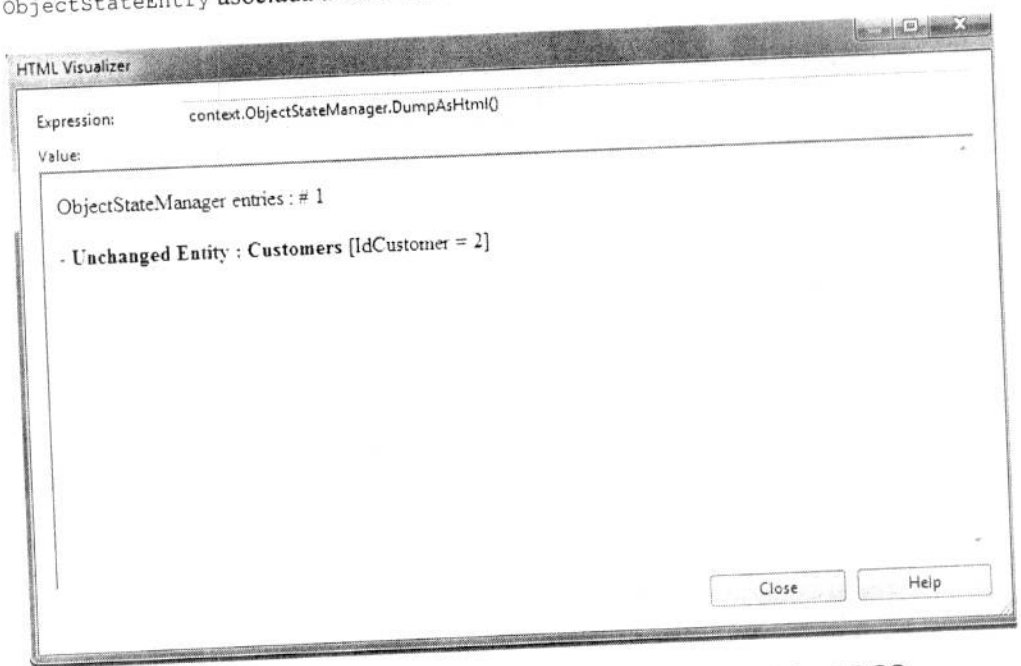


Figura 4.36.- Estado de ObjectStateManager y entidades POCO

Vayamos un paso más adelante y cambiemos el código anterior para realizar una modificación de la propiedad `LastName` de nuestra entidad `Customer`.

```
using (EF4EXEntities context = new EF4EXEntities())
{
    Customer customer = context.Customers.FirstOrDefault();
    customer.LastName = "Fernandez Vizoso";
}
```

Si pensamos en las entidades generadas por nuestra plantilla POCO, podremos ver como las propiedades primitivas como `LastName` solamente contienen los métodos `get` y `set` por defecto. Sin embargo, como muestra la figura 4.37, si revisamos nuestro OSM podemos ver como el contexto de trabajo ha sido capaz de "enterarse" del cambio de valor de esta propiedad.

```
public virtual string LastName
{
    get;
    set;
}
```

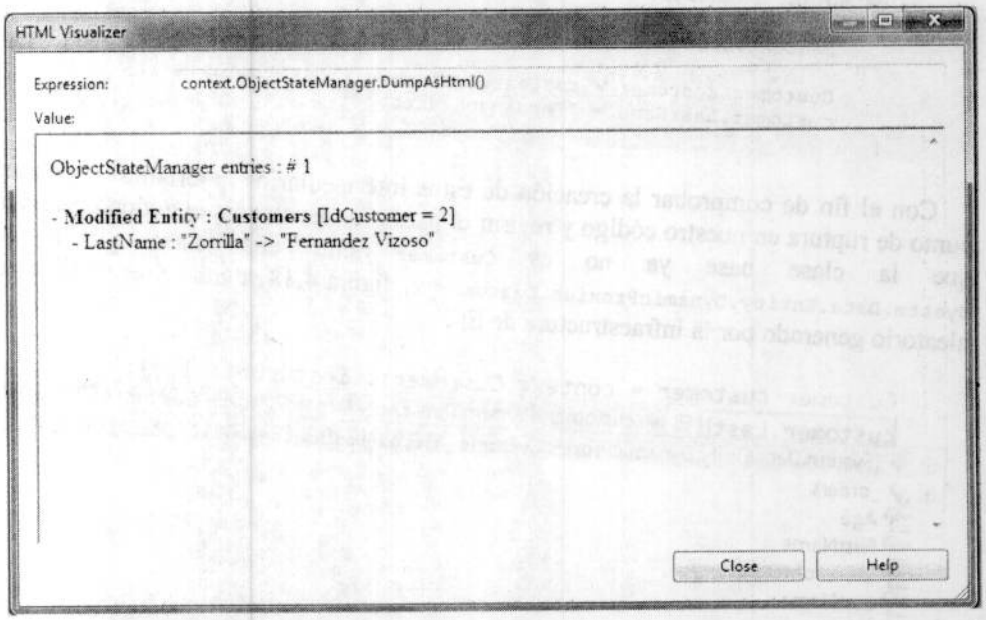


Figura 4.37.- Contenido del ObjectStateManager

¿Cómo es posible esta casuística si el método `set` de la propiedad anterior no notifica el cambio a nuestro contexto de trabajo, como por ejemplo en las clases prescriptivas?

La respuesta a esta "mágica" característica reside realmente en la capacidad de ADO.NET EF y las entidades POCO de generar "proxies" para las entidades consultadas. Un proxy, apoderado/intermediario, de una entidad POCO no es más que un tipo generado de forma dinámica que tiene a la entidad como clase base y que sobreescribe las propiedades virtuales para agregar las notificaciones de cambio correspondiente.

Si recuerda la propiedad `ObjectContextOptions`, de nuestros contextos de trabajo, en el cual podíamos decidir que tipo de comportamiento deseábamos para la opción de

mezcla `PreserveChanges` o bien para la carga perezosa, podrá recordar como dentro del tipo que la define disponemos de una propiedad, `ProxyCreationEnabled`.

```
public sealed class ObjectContextOptions
{
    public bool LazyLoadingEnabled { get; set; }

    public bool ProxyCreationEnabled { get; set; }

    public bool UseLegacyPreserveChangesBehavior { get; set; }
}
```

Como se podrá imaginar, esta propiedad nos permite habilitar o no la creación de los proxies de nuestras entidades consultadas en el contexto, por parte del contexto de trabajo. Para la plantilla de POCO, la única en la que realmente tienen sentido, esta opción está habilitada por defecto. Por supuesto, si usted lo desea, puede desabilitarla estableciendo a false esta característica:

```
using (EF4EXEntities context = new EF4EXEntities())
{
    context.ContextOptions.ProxyCreationEnabled = false;

    Customer customer = context.Customers.FirstOrDefault();
    customer.LastName = "Fernandez Vizoso";
}
```

Con el fin de comprobar la creación de estos intermediarios, podríamos poner un punto de ruptura en nuestro código y revisar el `Data Tip` asociado a la entidad para ver que la clase base ya no es `Customer` sino una clase similar a `System.Data.Entity.DynamicProxies.Customer_X`, figura 4.38, dónde X es un número aleatorio generado por la infraestructura de EF.

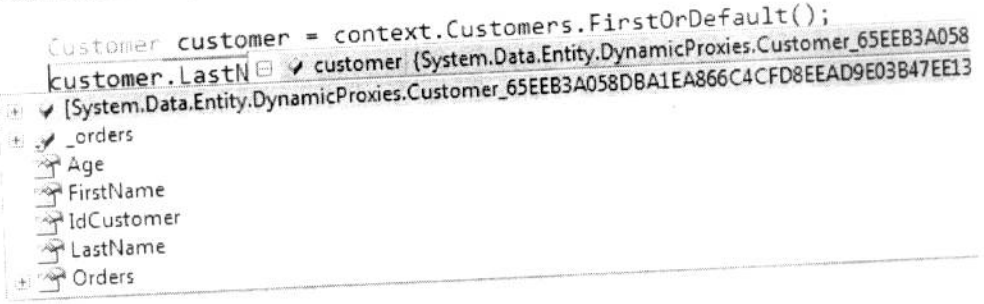


Figura 4.38.- Proxy de una entidad Customer

El objetivo de esta clase dinámica, como se comentó antes, es el de inyectar el código necesario para que estas entidades puedan funcionar dentro de la infraestructura. Las notificaciones de cambio o el manejo de las propiedades de navegación son ejemplos de los elementos que este tipo dinámico modifica.

Lógicamente, para ello, hay una serie de condiciones que se tienen que cumplir, entre ellas las siguientes:

- Las entidades no pueden estar selladas, para que se pueda heredar de ellas.
- Si necesitamos sobreescribir las propiedades, éstas deben de estar marcadas como virtuales.
- Puesto que se modificará el tipo de las propiedades de navegación, para cardinalidad múltiple, las mismas no deben de especificar el tipo, solamente un contrato.

Nota: Por supuesto, todas estas restricciones se cumplen por las entidades generadas por nuestra plantilla POCO, como cabía de esperar.

La posibilidad de usar entidades POCO, además de proporcionarnos de la forma más clara posible "ignorancia de la persistencia", nos permiten trabajar prácticamente con las mismas capacidades que las clases prescriptivas. Por supuesto, en un entorno distribuído, "sufren" las mismas deficiencias, puesto que cualquier gestión de los cambios de forma desconectada tiene que ser adhoc.

Valore el uso de estas entidades, siempre que necesite interoperabilidad con otras tecnologías o bien no quiera depender del código de seguimiento de las entidades STE, aunque sea generado de forma automáticamente junto a nuestras entidades.

Como nota adicional a estas entidades POCO, si probáramos el mismo ejemplo anterior, **code\ef4_ex_39**, con la opción de creación de objetos proxies deshabilitada, podrá comprobar como, aunque antes de hacer la llamada al método `SaveChanges`, el elemento `ObjectStateManager` no es consciente de la modificación de la propiedad (figura 4.39). Esta llamada produce la operación de modificación en la base de datos.

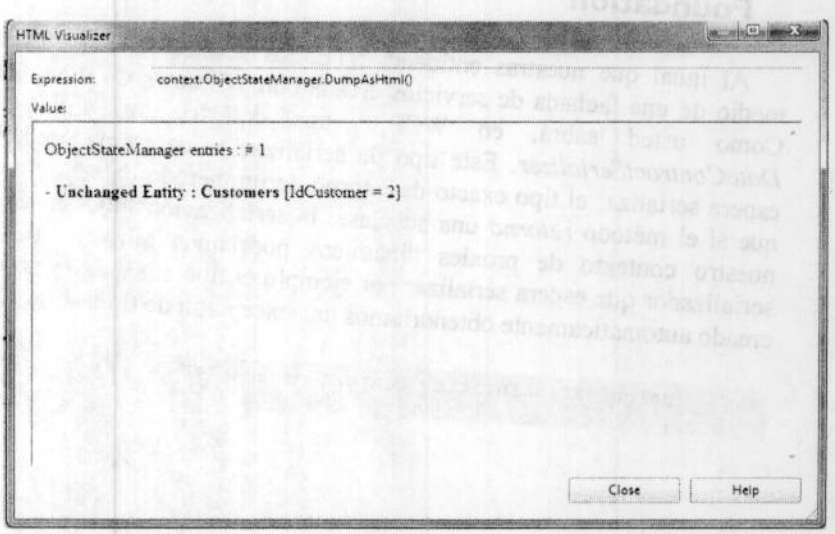


Figura 4.39.- Estado del ObjectStateManager antes de SaveChanges

Este comportamiento es posible gracias a una nueva capacidad del método `SaveChanges`, incorporada en EF 4.1, por la cual, antes de hacer las operaciones que se tengan que hacer en la base de datos, los `ObjectStateEntry` asociados al contexto de trabajo son sincronizados con los estados de las entidades, comprobando propiedad a propiedad si la misma se ha modificado y cual es el valor actual.

Nota: Este comportamiento, denominado generalmente como **matching**, podría llegar a tener un impacto de rendimiento. Piense en el esfuerzo extra de este trabajo para un número alto de entidades en juego. Por ello, el uso de objetos proxy suele ser más eficiente y por lo tanto más recomendable si usamos esta plantilla de entidades.

La utilización de esta característica puede habilitarse o deshabilitarse mediante una nueva sobrecarga del método `SaveChanges`, en virtud de la cual podemos incorporar el valor de un enumerado, `SaveOptions`, tal y como vemos en los siguientes fragmentos de código.

```
public virtual int SaveChanges(SaveOptions options);

public enum SaveOptions
{
    None = 0,
    AcceptAllChangesAfterSave = 1,
    DetectChangesBeforeSave = 2,
}
```

2.7.1.- POCO Proxies y Windows Communication Foundation

Al igual que nuestras entidades STE, los objetos POCO pueden exponerse por medio de una fachada de servicios creada con Windows Communication Foundation. Como usted sabrá, en WCF por defecto, se utiliza la serialización *DataContractSerializer*. Este tipo de serialización de forma predefinida únicamente espera serializar el tipo exacto de retorno de un método de servicio. Esto quiere decir que si el método *retorna* una subclase, la serialización fallaría. Si ponemos esto en nuestro contexto de proxies dinámicos podríamos afirmar que si utilizamos un serializador que espera serializar por ejemplo el tipo `Customer` y serializamos el proxy creado automaticamente obtendríamos una excepción de tipo `SerializationException`.

```
using (EF4EXEntities context = new EF4EXEntities())
{
```

```
        context.ContextOptions.ProxyCreationEnabled = true;

        Customer customer = context.Customers.FirstOrDefault();
        customer.LastName = "Fernandez Vizoso";

        DataContractSerializer serializer = new
    DataContractSerializer(typeof(Customer));

        MemoryStream stream = new MemoryStream();

        //next line raise SerializationException
        serializer.WriteObject(stream, customer);

    }
```

Para resolver este problema, WCF dispone de una funcionalidad conocida como el "polimorfismo de los contratos de datos", gracias a la cual a nuestro serializador le podemos indicar los distintos elementos de la jerarquía que debe de conocer a la hora de serializar.

Para indicar estos "tipos a conocer", en Windows Communication Foundation disponemos de los atributos `KnownType` y `ServiceKnownType`. Por desgracia, puesto que el tipo se genera de forma dinámica y no conocemos su nombre, estos atributos no pueden utilizarse.

Como novedad en .NET 4.1, además de estos dos atributos disponemos de un nuevo tipo conocido como `DataContractResolver`, tipo que nos permitirá indicar como resolver los tipos de forma dinámica. El equipo de trabajo de EF creó para nosotros una subclase de este tipo, conocida como `ProxyDataContractResolver` que justamente nos permitirá realizar esta tarea.

Para ver su uso, modificaremos el código anterior, haciendo uso de una de las múltiples sobrecargas de `DataContractSerializer`:

```
    using (EF4EXEntities context = new EF4EXEntities())
    {
        context.ContextOptions.ProxyCreationEnabled = true;

        Customer customer = context.Customers.FirstOrDefault();
        customer.LastName = "Fernandez Vizoso";
        ProxyDataContractResolver proxyResolver = new
    ProxyDataContractResolver();

        DataContractSerializer serializer = new
    DataContractSerializer(typeof(Customer),null,Int32.MaxValue,false,tru
    e,null,proxyResolver);

        MemoryStream stream = new MemoryStream();

        serializer.WriteObject(stream, customer);

    }
```

Como se puede comprobar ejecutando el código anterior, gracias a nuestra instancia de `ProxyDataContractResolver`, el proceso de serialización funciona ahora correctamente.

La pregunta ahora es, ¿como podemos hacer en **WCF** que el serializador use esta sobrecarga?. La respuesta es que se consigue por medio de una extensión de nuestros comportamientos de operación, como podemos ver a continuación.

```
public class ProxyOperationBehavior : Attribute, IOperationBehavior
{
    public void AddBindingParameters(
                    OperationDescription description,
                    BindingParameterCollection parameters)

    {
    }

    public void ApplyClientBehavior(
                    OperationDescription description,
                    ClientOperation proxy)

    {
        DataContractSerializerOperationBehavior
              dataContractSerializerOperationBehavior =

description.Behaviors.Find<DataContractSerializerOperationBehavior>()
;

    dataContractSerializerOperationBehavior.DataContractResolver =
              new ProxyDataContractResolver();
    }

    public void ApplyDispatchBehavior(
                    OperationDescription description,
                    DispatchOperation dispatch)

    {
          DataContractSerializerOperationBehavior
            dataContractSerializerOperationBehavior =

description.Behaviors.Find<DataContractSerializerOperationBehavior>()
;

dataContractSerializerOperationBehavior.DataContractResolver =
              new ProxyDataContractResolver();
    }

    public void Validate(OperationDescription description)
    {
    }
}
```

Una vez creado este nuevo comportamiento de operación, solamente tendríamos que establecernos en nuestras operaciones de sevicio, como por ejemplo se muestra en el siguiente fragmento de código.

```
[ServiceContract()]
public interface IWCFSample
{
    [OperationContract()]
```

```
    Customer GetCustomer();
}

public class WCFSample : IWCFSample
{
    #region IWCFSample Members

    [ProxyOperationBehavior()]
    public Customer GetCustomer()
    {
        using (EF4EXEntities context = new EF4EXEntities())
        {
            return context.Customers.First();
        }
    }

    #endregion
}
```

CAPÍTULO 5

EF 4.1 En el mundo real

1.- INTRODUCCIÓN

Hasta ahora, en los capítulos anteriores de este libro hemos visto las distintas partes que conforman a ADO.NET Entity Framework 4.1. Desde la creación de modelos de dominio hasta los artefactos de generación y los distintos tipos de entidades con los que podemos trabajar.

La idea del presente capítulo es tratar de mostrar las posibilidades que EF nos permite en una solución real. Para ello, nos basaremos en un proyecto de Microsoft DPE España en la que los autores hemos tenido la suerte de participar: **Microsoft NLayer App**. Su código está alojado en el portal de proyectos Open Source de Microsoft, **Codeplex**.

Este proyecto nació con la idea de presentar un ejemplo real y completo de uso de diferentes tecnologías relacionadas con .NET 4.1, así como diferentes patrones de uso común en arquitecturas guiadas por modelos de dominio

Nota: El código del proyecto de Microsoft NLayer App puede ser descargado directamente desde la dirección **http://microsoftnlayerapp.codeplex.com/**. Además, en esta misma dirección podrá encontrar foros de discusión, documentación de ayuda y los distintos ChangeSet del control de código fuente asociado para los distintos sabores de la solución.

Además del código esta iniciativa dispone de un libro blanco de arquitectura, editado por Krasis Press para Microsoft, que puede descargar de forma gratuita en diferentes formatos (PDF, XPS, .mobi, .epub). Para acceder a la descarga de este libro puede navegar a la página principal de DPE Arquitectura: **http://msdn.microsoft.com/es-es/architecture/default.aspx**

Aunque dentro de esta iniciativa de Microsoft también podemos encontrar un libro blanco de arquitectura, éste no profundiza en lo relacionado con la capa de persistencia y Entity Framework 4.1, algo que sí haremos en este capítulo. No obstante, sería recomendable la lectura de los distintos componentes arquitectónicos en juego ya que, sin lugar a dudas, le permitiría situarse mucho mejor en la lectura de los siguientes contenidos de este capítulo.

Empezaremos nuestro trabajo poniendo el diagrama arquitectónico en el que nos basaremos para construir las diferentes partes de nuestra capa de persistencia y los componentes de Entity Framework que utilizaremos.

Arquitectura N-Capas con Orientación al Dominio

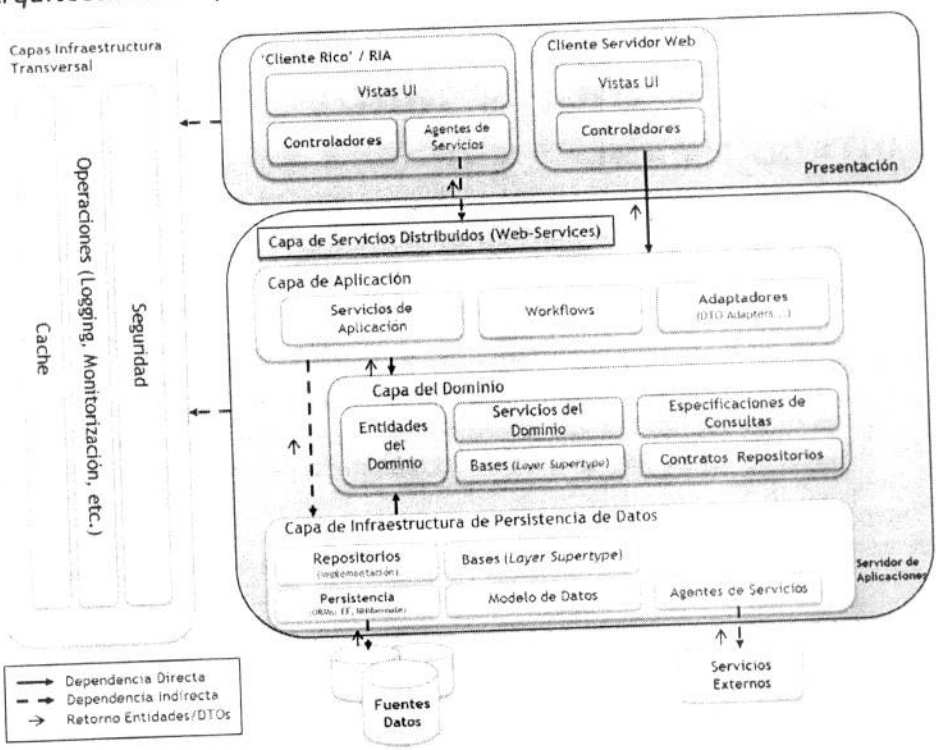


Figura 5.1.- Diagrama arquitectónico de nuestra aplicación de ejemplo

2.- EL DOMINIO, LAS ENTIDADES Y EF 4

El dominio es, sin duda alguna, la parte más fundamental de nuestro trabajo. La selección de las entidades, su disposición y restricciones deben ser elementos en los que tenemos que poner toda nuestra atención.

En el capítulo anterior hicimos bastante hincapié en conceptos como la ignorancia de la persistencia y las ventajas que nos aporta aislar la definición de nuestro modelo de una tecnología en concreto. Sin embargo, en todos los ejemplos realizados, la disposición de nuestras entidades se encontraba en el mismo nivel lógico que nuestros contextos de trabajo, APIs de EF etc...

Nota: No en pocas ocasiones se suele asociar el concepto de ignorancia de la persistencia con la capacidad de cambiar el motor de persistencia de una forma rápida y/o transparente. En realidad esto es simplemente una consecuencia, no el propósito que se busca puesto que sólo en raras, muy raras, ocasiones uno decide migrar un motor de persistencia funcionando hacia otro. La ignorancia de la persistencia persigue "solamente" aplicar principios de diseño como la separación de responsabilidades y el aislamiento de dependencias.

A lo largo de este punto, veremos cómo separar nuestras entidades de los modelos de EDM hacia nuestras capas de dominio, y como empezar a poner en práctica distintos patrones y principios de diseño, algunos de ellos, comentados en anterioridad y otros de los que discutiremos. Para este trabajo, partiremos de una solución, **code\ef4_sample1**, con una estructura determinada, figura 5.2, en la que iremos agregando distintos elementos.

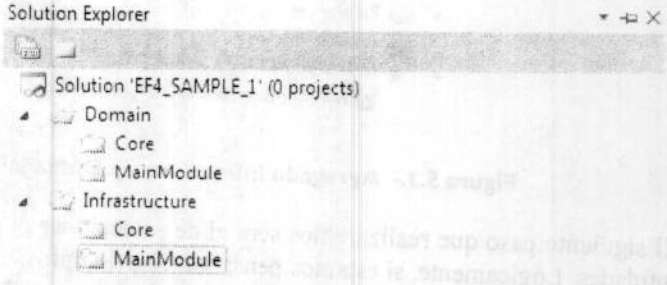


Figura 5.2.- Estructura básica de la solución

Por ahora, hasta la llegada de EF Feature Pack en el Q1 del 2011, para trabajar con ADO.NET Entity Framework, siempre debemos partir de un modelo de EDM. Este hecho, de entrada, no nos permite definir un conjunto de entidades en nuestro dominio de forma independiente a un modelo de EF. Por esta razón, por ahora, para generar nuestras entidades debemos empezar siempre por la definición de un modelo EDM, :-(

Para nuestro ejemplo, realizaremos un pequeño módulo, las típicas entidades del capítulo anterior, y, a partir de él, veremos cómo empezar. Si se fija, en la estructura de nuestra solución de ejemplo, hemos hecho dos divisiones fundamentales: una carpeta de solución para los proyectos a nivel de dominio, y otra carpeta para aquellos a nivel de infraestructura. Además de estas carpetas hemos incluido en cada una de ellas otra

subdivisión, una de *core* y otra para un módulo particular que hemos llamado *MainModule*. Situaremos por lo tanto un nuevo proyecto de biblioteca de clases dentro de *Infrastructure\MainModule* que llamaremos *Infrastructure.Data.MainModule* y agregaremos en él un modelo de EDM con las entidades *Customer*, *Order* y *OrderDetail* que ya conocerá de los distintos ejemplos pasados.

Nota: En el capítulo 3 hablamos de la división de modelos y de cómo trabajar con lo "grande". De hecho, se expusieron distintas técnicas de "bounded context" como *Separte ways*, *customer suplier* o *shared kernel*. Piense que nuestra separación por módulos, como *MainModule*, no es más que un ejemplo de *Separate Ways*, es decir, una separación de módulos independientes dentro de una misma solución que no comparten entidades ni dependen unos de los otros.

En la siguiente imagen, figura 5.3, puede ver este proyecto incorporado a nuestra estructura de la solución.

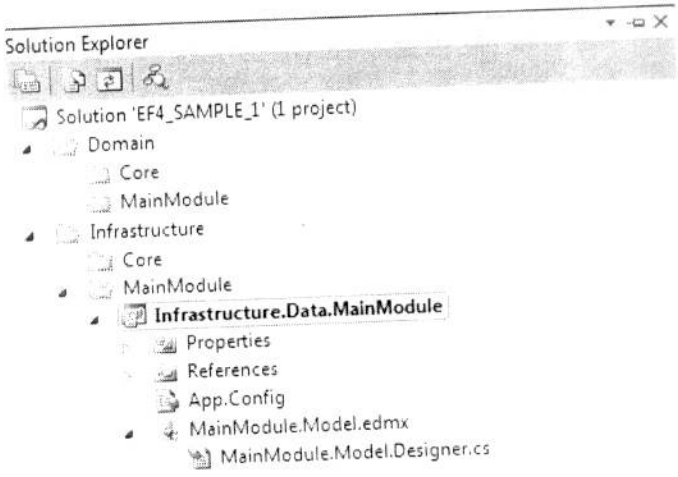


Figura 5.3.- Agregado Infrastructure.Data.MainModule

El siguiente paso que realizaremos será el de seleccionar el artefacto de generación de entidades. Lógicamente, si estamos pensando en principios como el de ignorancia de la persistencia, la utilización de clases prescriptivas queda descartada. Para nuestro caso (tal y como está también en el ejemplo completo del proyecto de Codeplex) seleccionaremos el artefacto de *Selft Tracking Entities*, de la misma forma que hemos hecho ya en el capítulo anterior.

Como ya sabrá a estas alturas, al agregar este artefacto dentro de nuestro proyecto se agregarán dos nuevas plantillas de T4, una correspondiente a la generación del contexto de trabajo, nuestro Unit of Work, y otra correspondiente a la generación de entidades. Nuestro objetivo será ver cómo podemos mover la generación de estas entidades a un nuevo ensamblado en el dominio. Para ello volveremos a agregar una

biblioteca de clases a nuestra solución, esta vez dentro de la carpeta *domain\mainmodule* que llamaremos *Domain.MainModule.Entities*. Revise la figura 5.4, que contiene ya los dos proyectos de biblioteca de clases agregados a nuestra solución:

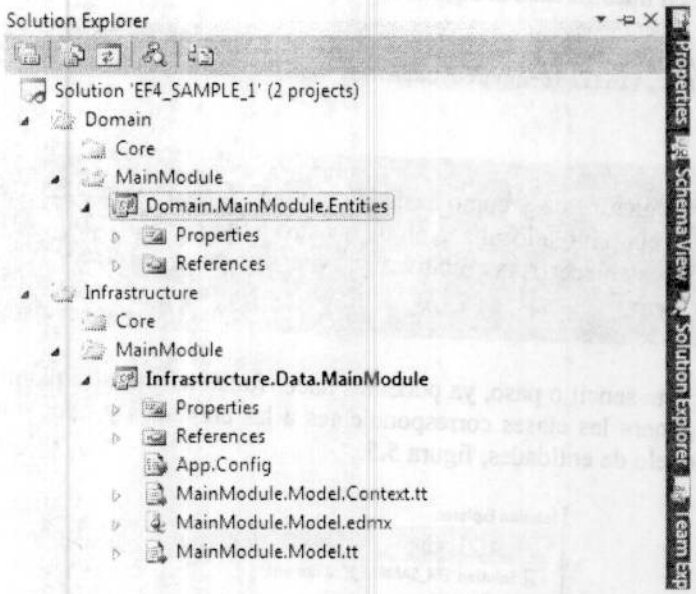


Figura 5.4.- Agregado Domain.MainModule.Entities

Para llevar nuestra plantilla de entidades del proyecto de infraestructura a nuestro proyecto de dominio solamente tendremos que copiar y pegar de uno a otro proyecto el elemento *MainModule.Model.tt*.

Nota: Una alternativa al proceso de mover la plantilla de generación de entidades es usar las capacidades de archivos enlazados de Visual Studio. Sin embargo, en nuestra opinión, fundamentada por la experiencia, suele dar más 'preocupaciones y molestias' que la forma que hemos recomendado.

Una vez que ya hemos movido la plantilla de generación, si la *ejecutamos* mediante su opción *Run Custom Tool* o simplemente guardándola, podremos observar un error dentro de Visual Studio con el siguiente mensaje de excepción:

```
Running transformation: System.Reflection.TargetInvocationException:
Exception has been thrown by the target of an invocation. --->
System.IO.FileNotFoundException: Unable to locate file
```

Este error se produce porque las plantillas de T4 para nuestras entidades (aquí es independiente de si son *STE*, *POCO* o prescriptivas) lógicamente necesitan

inspeccionar la información contenida dentro del fichero **edmx,** los distintos espacios CSDL, SSDL y MSL. El *path* hacia el fichero **edmx** a inspeccionar, el modelo con el que se han creado estas plantillas al fin y al cabo, se encuentra definido al comienzo del código de la plantilla, concretamente en la siguiente línea:

```
string inputFile = @"MainModule.Model.edmx";
```

Lógicamente, como hemos movido la plantilla de entidades, no se puede encontrar el fichero indicado en esta línea, para arreglarlo, solamente tenemos que poner el camino correcto, en nuestro caso el siguiente:

```
string inputFile =
@"..\Infrastructure.Data.MainModule\MainModule.Model.edmx";
```

Nota: Lógicamente y como casi siempre, NO sería recomendable establecer una ruta concreta para la localización de nuestro fichero edmx. Como en nuestro caso, es mejor establecer rutas relativas y evitar así posibles errores si trabajamos en equipo.

Con este sencillo paso, ya podemos hacer funcionar nuestra plantilla, y ver como la misma genera las clases correspondientes a las entidades y tipos complejos definidos en el modelo de entidades, figura 5.5.

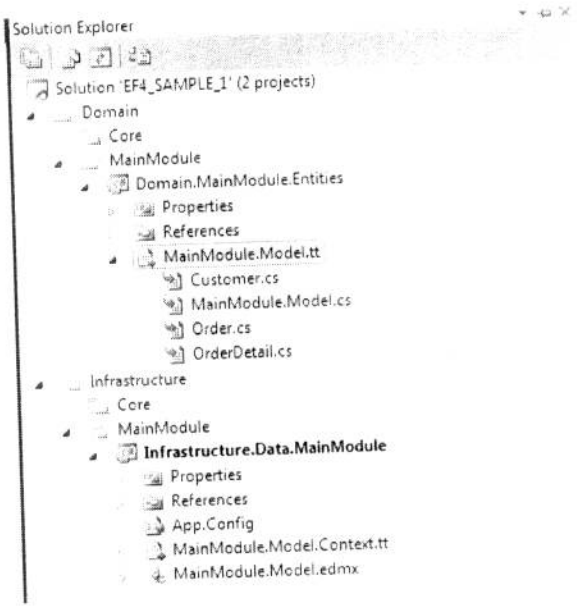


Figura 5.5.- Modificada la plantilla de generación de entidades

Si abrimos alguno de los ficheros de entidades generados, podremos observar como las entidades están decoradas con los atributos DataContract y KnownType, atributos utilizados para la serialización de contratos de datos con DataContractSerializer.

```
[DataContract(IsReference = true)] [KnownType(typeof(Order))]
public partial class Customer: IObjectWithChangeTracker,
INotifyPropertyChanged
{
    ...
```

Puesto que, por defecto, los proyectos de bibliotecas de clases como el que hemos creado nosotros no incorporan como referencia el ensamblado *System.Runtime.Serialization*, ensamblado en el que están definidos los atributos anteriores, debemos de agregar una referencia al mismo, para que compile correctamente.

Nota: Aunque no se va a hacer por no desviarnos del tema que nos ocupa, las plantillas de T4, de entidades y del contexto de trabajo, también se suelen tocar con la finalidad de agregar/modificar distintos elementos. Un ejemplo podría ser la inclusión de los atributos *GeneratedCode* o *ExcludeFromCodeCoverage* para que este código autogenerado no nos influya en nuestras métricas de calidad como tasa de cobertura de código

Si volvemos a nuestro proyecto de infraestructura, dónde habíamos incluido nuestro fichero de EDM, podremos comprobar como al intentar compilar se nos presentan multitud de errores, todos relacionados con el mismo problema. Como sabemos, para cada modelo de entidades la plantilla de STE genera un contexto de trabajo dónde, además de todas las funcionalidades inherentes a él, disponemos de propiedades para cada uno de los **EntitySets** definidos. Lógicamente, estas propiedades están definidas por las entidades, las cuales hemos llevado hacia otro proyecto y por lo tanto no conoce. Podríamos pensar de entrada que sería suficiente con agregar a este proyecto de infraestructura una referencia a nuestro proyecto de entidades. Pero aunque este es un paso necesario no es suficiente, ya que el código de las entidades generadas no hace uso del *using* necesario que permita cualificar a las mismas.

Para resolver este problema la plantilla de generación pone a nuestra disposición un método llamado ***WriteHeader***. Este método, se encarga de escribir en los ficheros de salida las cabeceras de los mismos, cabeceras que incluyen los *using* y la cualificación del *espacio de nombres*. En la figura 5.6 puede ver la firma e *intellisense* de este método. Como observará, el segundo parámetro nos ofrece la posibilidad de indicar qué nuevos *using* deseamos agregar al código generado.

```
if (container == null)
{
    return "// No EntityContainer exists in the model, so no code was generated";
}

WriteHeader(fileManager,);
 void WriteHeader(Host, FrameworkTemplateFileManager fileManager, string[] extraUsings)

#>
<#=Accessibility.ForType(container)#> partial class <#=code.Escape(container)#> : ObjectContext
```

Figura 5.6.- Firma del método WriteHeader.

Nota: Por supuesto, si usted no se siente cómodo con el trabajo con las plantillas T4 o desconoce las mismas le recomendamos desde aquí que revise el apéndice sobre el tema incluido en este mismo libro. Por defecto Visual Studio no tiene soporte de "code coloring" e "intellisense" en las plantillas T4. Puede usted disponer, al igual que nosotros, de ello haciendo uso de una extensión gratuita llamada **Tangible Editor**. Esta extensión está disponible directamente en el *Extension Manager* de Visual Studio 2010.

El siguiente fragmento contiene la modificación que debemos de realizar a nuestro método WriteHeader.

```
WriteHeader(fileManager,"Domain.MainModule.Entities");
```

Una vez, hecho esto, el código del contexto de trabajo generado ya contendrá nuestro nuevo *using* y por lo tanto las entidades definidas en nuestro anterior proyecto ya se localizan correctamente.

```
using System.ComponentModel;
using System.Data.Common;
using System.Data.EntityClient;
using System.Data.Metadata.Edm;
using System.Data.Objects.DataClasses;
using System.Data.Objects;
using System.Data;
using System.Diagnostics;
using System.Globalization;
using System.Linq;
using Domain.MainModule.Entities;

namespace Infrastructure.Data.MainModule
{
    public partial class MainModuleUnitOfWork : ObjectContext
    {
```

2.1.1.- Definiendo las abstracciones

Llegados a este punto, en el que hemos separado en distintos proyectos la generación de nuestras entidades y nuestro contexto de trabajo, empezaremos a poner de manifiesto alguno de los patrones más comunes en modelos de dominio.

El primero que trataremos será el patrón **Unit Of Work**, figura 5.7, definido por *Martin Fowler* en su obra de referencia "Patterns Of Enterprise Application Architecture". Una unidad de trabajo, de forma coloquial, podría decirse que se encarga de mantener una lista de aquellas entidades afectadas en una transacción de negocio, de tal forma que podamos enviar en un momento dado los cambios efectuados sobre los mismos a una base de datos.

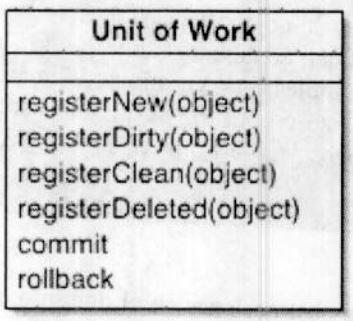


Figura 5.7.- Patrón Unit of Work

Si lo piensa con detenimiento, los contextos de trabajo de ADO.NET EF son unidades de trabajo y, el `ObjectStateManager` la pieza dentro de los contextos que nos permite realizar esta tarea.

Los métodos típicos de este patrón como `registerNew` o `registerDelete` podrían asociarse con los métodos `AddObject` y `DeleteObject` de nuestros `ObjectSet`. El caso de `registerDirty` no tiene un elemento asociado puesto que generalmente se hace de forma transparente, sin necesidad de una llamada a ningún método de registro.

El uso de este patrón, UoW, es muy común dentro de nuestros dominios, puesto que, en definitiva, es la forma que tenemos para decidir cuándo comprometer los cambios realizados en un proceso de negocio. Lógicamente, imponer dentro del dominio que nuestras unidades de trabajo sean contextos de trabajo de Entity Framework sería un error. Estaríamos cargándonos de un plumazo el concepto de ignorancia de la persistencia, con todo lo que esto conlleva, además de otros principios de desarrollo fundamentales. Con la finalidad de superar esto, modificaremos nuestra solución incluyendo dentro de las carpetas *domain\core* una biblioteca de clases, *Domain.Core*, dónde trataremos de plasmar este patrón, junto con alguno a mayores que veremos a continuación, **code\ef4_sample2**.

Empezaremos en este nuevo proyecto incluyendo nuestra definición de una unidad de trabajo, vea el siguiente fragmento de código:

```
public interface IUnitOfWork:  IDisposable
{
    void Commit();

    void CommitAndRefreshChanges();

    void RollbackChanges();

    void RegisterChanges<TEntity>(TEntity item)
            where TEntity : class, IObjectWithChangeTracker;
}
```

Como puede observar, este contrato nos permite definir las operaciones normales dentro de una unidad de trabajo, comprometer los cambios, deshacerlos y hacer un registro de operaciones. El motivo principal de la creación de este nuevo contrato viene por el hecho de hacer prevalecer nuestra ignorancia de la persistencia, aportando para nuestras unidades de trabajo un contrato *ignorante* de la tecnología a utilizar.

Nota: Dentro de este contrato, como puede observar, se dispone de un método llamado **RegisterChanges**. En no pocas ocasiones, se ha discutido si este método estaba ligado a EF, en realidad al método *ApplyChanges* de nuestras *Self Tracking Entities*. Aunque son entendibles estas cuestiones, en realidad este método no está creado ni nombrado con este propósito, ya que el mismo podría estar asociado, por ejemplo, con los métodos *Merge* y *SaveOrUpdate* de NHibernate.

La siguiente de nuestras abstracciones en el dominio, a incluir también en nuestro proyecto Domain.Core es la de los **repositorios**. Un repositorio, *Repository*, es una de las formas bien documentadas para trabajar con una fuente de datos. Al igual que con UoW, otra vez *Martin Fowler* en su libro PoEAA describe un repositorio de la siguiente forma:

"*Un repositorio realiza las tareas de intermediario entre las capas de modelo de dominio y mapeo de datos, actuando de forma similar a una colección en memoria de objetos del dominio. Los objetos cliente construyen de forma declarativa consultas y las envían a los repositorios para que las satisfagan. Conceptualmente, un repositorio encapsula a un conjunto de objetos almacenados en la base de datos y las operaciones que sobre ellos pueden realizarse, proveyendo de una forma más cercana a la orientación a objetos de la vista de la capa de persistencia. Los repositorios, también soportan el objetivo de separar claramente y en una dirección la dependencia entre el dominio de trabajo y el mapeo o asignación de los datos*".

Este patrón, es uno de los más habituales hoy en día, sobre todo si pensamos en *Domain Driven Design*, puesto que nos permite de una forma sencilla conseguirque

nuestras capas de datos sean '*testables*', y trabajar de una forma más simétrica a la orientación a objetos con nuestros modelos relaciones .

Así pues, para cada tipo de objeto que necesite acceso global (normalmente por cada Entidad raíz de un *Agregado*), se debe crear un objeto (Repositorio) que proporcione la apariencia de una colección en memoria de todos los objetos de ese tipo.

Se debe establecer acceso mediante un interfaz bien conocido, proporcionar métodos para consultar, añadir, modificar y eliminar objetos, que realmente encapsularán la inserción o eliminación de datos en el almacén de datos. Proporcionar métodos que seleccionen objetos basándose en ciertos criterios de selección y devuelvan objetos o colecciones de objetos instanciados (entidades del dominio) con los valores de dicho criterio, de forma que encapsule el almacén real (base de datos, etc.) y la tecnología base de consulta.

En una solución más o menos grande, la mayoría de los repositorios contienen operaciones comunes. Además de las típicas de mantenimiento como agregar o eliminar, es habitual que en todos se disponga de operaciones para recuperar por un criterio, de forma ordenada, paginada etc. Teniendo en cuenta esto, se podría hacer un contrato de repositorio común que englobara las operaciones habituales de un repositorio. A continuación se puede ver el fragmento de código que define a un posible contrato de repositorio:

```
public interface IRepository<TEntity>
        where TEntity : class
{
   IUnitOfWork UnitOfWork { get; }

   void Add(TEntity item);

   void Remove(TEntity item);

   void RegisterItem(TEntity item);

   void Modify(TEntity item);

   IEnumerable<TEntity> GetAll();
}
```

Nota: Basar los repositorios en un contrato común es discutible puesto que estamos imponiendo a todos los repositorios una serie de atribuciones que no tendrían por qué tener, ya que, por ejemplo, no en todos los repositorios tendría que poder buscarse de forma paginada. La inclusión de un contrato común tiene que ser una decisión meditada en la que se valore la pureza de las responsabilidades con respecto a una mejora en productividad que podría obtenerse con la inclusión de este contrato común.

Como se puede observar en la figura siguiente, tendremos una clase de *Repository* por cada entidad lógica de datos (entidad principal o también llamadas en DDD como *AGGREGATE roots*), que puede estar representada/persistida en la base de datos por una

o más tablas, pero sólo uno de los tipos de 'objeto' será el tipo raíz por el que se canalizará.

Relación entre clases de Repositorios y Entidades

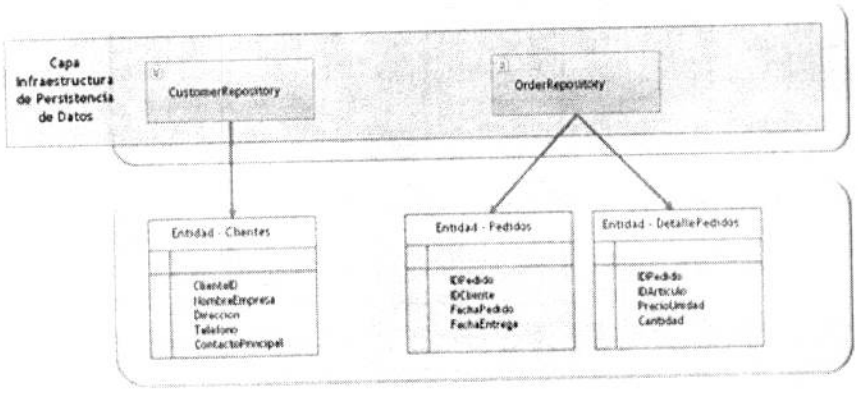


Figura 5.8.- Relaciones entre repositorios y entidades

Nota: Decidir en un modelo de dominio cual es una entidad raíz y cual no es un ejercicio de "practica", que lógicamente dependerá de lo que se esté intentando modelar. Le recomendamos desde aquí la lectura de "Domain Driven Design" de Eric Evans.

Para nuestro ejemplo, recuerde que tan solo disponíamos de las entidades *Customer*, *Order* y *OrderDetail*. Definiremos dos repositorios llamados *CustomerRepository* y *OrderRepository*. Su disposición se situará en un nuevo proyecto dentro del dominio llamado Domain.MainModule, vea la figura 5.9.

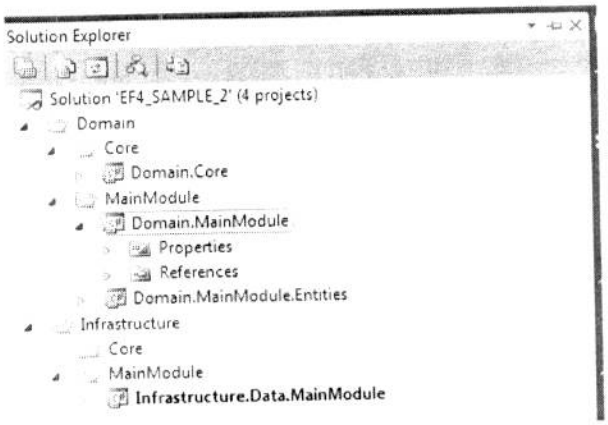


Figura 5.9.- Creación del proyecto Domain.MainModule

En este proyecto empezaremos por crear los contratos de nuestros repositorios. Fíjese en que el término "contrato" es muy importante ya que lo único que necesitamos es saber las operaciones que nos permiten realizar los mismos. La implementación no es relevante en este punto.

> **Nota:** Aún a riesgo de parecer repetitivo le recomendamos desde aquí la lectura de la guía de arquitectura comentada al comienzo de este libro. En esta guía, se exponen distintos principios de diseño como la inyección de dependencias, que le servirán de utilidad para entender mejor la forma de trabajo y sobre todo los beneficios que podrá obtener.

El primero de los contratos de nuestros repositorios será el de *Customer*, que tendrá un aspecto similar al presentado a continuación:

```
public interface ICustomerRepository
        :IRepository<Customer>
{
    IEnumerable<Customer> GetCustomersByFirstName(string firstName);
}
```

Por supuesto, nuestro contrato de repositorio implementa el contrato común que se decidió crear para todos los elementos, *IRepository*, para lo cual hemos necesitado agregar una referencia a *Domain.Core*. Lógicamente, agregar una referencia a *Domain.MainModule.Entities* también es necesario para poder referenciar a las entidades.

El caso de nuestro repositorio de *Order* es igual de sencillo A continuación presentamos el fragmento de código con la definición del mismo.

```
public interface IOrderRepository
        :IRepository<Order>
{
   IEnumerable<Order> GetOrderWithDetails(int orderIdentifier);

   IEnumerable<Order> GetOrdersBetweenDates(DateTime fromDate,
DateTime toDate);

   IEnumerable<Order> GetOrderByCustomerId(int customerIdentifier);

}
```

2.1.2.- Estableciendo la infraestructura

En el punto anterior se prepararon los contratos para dos elementos fundamentales en arquitecturas orientadas al dominio, como son las unidades de trabajo y los

repositorios. A continuación veremos cómo definir nuestros repositorios concretos con ADO.NET EF, y cómo hacer que nuestros contextos de trabajo sean implementaciones del contrato *IUnitOfWork* anterior.

Tal y como ya sabemos a estas alturas, el disponer de plantillas de Text Templating para la generación de los distintos elementos nos permite modificar el código de los distintos elementos de EF de una forma rápida y sencilla. Gracias a esto, hacer que los contextos de trabajo sean implementaciones de nuestro contrato *IUnitOfWork* es una tarea sencilla que podremos hacer en pocos pasos, modificando la plantilla adecuada, tal y como se muestra a continuación.

1. En *Infrastructure.Data.MainModule* tendremos que agregar una referencia hacia el ensamblado dónde se define nuestro contrato *IUnitOfWork*, que en nuestro caso es *Domain.Core*.

2. Una vez agregada la referencia anterior, modificaremos nuestra plantilla *MainModule.Model.Context.tt* para que incluya dentro del código generado el "*using Domain.Core*". Para ello, como hemos hecho anteriormente, solamente hay que agregar un nuevo parámetro al método *WriteHeader* definido en la plantilla.

```
WriteHeader(fileManager,"Domain.MainModule.Entities","Domain.Core");
```

3. El siguiente paso, consistirá en localizar el comienzo de la definición de nuestra clase de contexto de trabajo de EF y establecer que la misma, además de heredar de *ObjectContext*, también implementa nuestra interfaz *IUnitOfWork*.

```
<#=Accessibility.ForType(container)#>          partial          class
<#=code.Escape(container)#> : ObjectContext,IUnitOfWork
{
```

4. Lógicamente, dentro de esta clase, tendremos que incluir la implementación de los métodos definidos en el nuevo contrato que implementa. Por lo tanto, dentro del cuerpo de esta clase, en la plantilla T4, tendremos que incluir las siguientes líneas.

```
public void Commit()
{
    this.SaveChanges();
}
public void RollbackChanges()
{
   //Refresh context and override changes

   IEnumerable<object> itemsToRefresh = base.ObjectStateManager
    .GetObjectStateEntries(EntityState.Modified)
    .Where(ose => !ose.IsRelationship && ose.Entity != null)

   .Select(ose => ose.Entity);
    base.Refresh(RefreshMode.StoreWins, itemsToRefresh);
}
```

```
public void CommitAndRefreshChanges()
{
    try
    {
        this.SaveChanges();
    }
    catch (OptimisticConcurrencyException ex)
    {
        this.Refresh(RefreshMode.ClientWins, ex.StateEntries.Select(e
=> e.Entity));
        this.SaveChanges();
    }
}
public void RegisterChanges<TEntity>(TEntity item) where
TEntity:class,IObjectWithChangeTracker
{

    this.CreateObjectSet<TEntity>()
        .ApplyChanges(item);
}
```

Una vez hecho esto, ya disponemos de un contexto de trabajo de Entity Framework 4.1 que implementa nuestra interfaz *IUnitOfWork*, y por lo tanto es válido para ser inyectada cuando se necesite.

El siguiente paso que llevaremos a cabo será el de implementar los distintos repositorios definidos anteriormente: *ICustomerRepository* y *IOrderRepository*. Para esta tarea podríamos pensar en implementar un repositorio genérico, puesto que si ya hemos definido un contrato base para todos los repositorios, entonces tendría sentido crear un repositorio genérico capaz de dar implementación a los métodos comunes.

Partiremos del nuestro ejemplo anterior e iremos en la solución **code\ef4_sample3** completando nuestras tareas. Lo primero, será la creación de un nuevo proyecto, denominado *Infrastructure.Data.Core*, que tiene como objetivo la definición de los distintos elementos comunes a cualquier módulo con respecto a la persistencia. En este módulo crearemos nuestra clase genérica *Repository*, la cual intentará dar implementación al contrato común de los repositorios para cualquier agregado raíz.

Inicialmente, nuestra clase genérica de repositorios tendrá un aspecto similar al siguiente:

```
public class Repository<TEntity> :IRepository<TEntity>
where TEntity:class
{
        #region IRepository<TEntity> Members

        public IUnitOfWork UnitOfWork
        {
            get { throw new NotImplementedException(); }
        }

        public void Add(TEntity item)
        {
            throw new NotImplementedException();
        }

        public void Remove(TEntity item)
```

```
        {
            throw new NotImplementedException();
        }

        public void RegisterItem(TEntity item)
        {
            throw new NotImplementedException();
        }

        public void Modify(TEntity item)
        {
            throw new NotImplementedException();
        }

        public IEnumerable<TEntity> GetAll()
        {
            throw new NotImplementedException();
        }

        #endregion
    }
```

Si nos preguntáramos cómo realizar ahora la implementación de este repositorio, seguramente tendríamos serias dudas. Tanto *IRepository<TEntity>* como *IUnitOfWork*, los dos elementos de los que depende esta clase, fueron pensados para ser ignorantes de la persistencia. Por lo tanto, intentar gracias a ellos hacer los diferentes trabajos que tenemos que hacer contra Entity Framework resulta imposible.

Para solventar este problema, crearemos una pequeña abstracción más en este mismo proyecto por medio de un nuevo contrato que llamaremos ***IQueryableUnitOfWork***. Este nuevo contrato tiene como objetivo definir una unidad de trabajo que sea a mayores un elemento que conozca EF, hemos asociado el nombre *Queryable* con L2E, aunque en realidad como ya sabrá esto no tiene por qué ser así.

```
public interface IQueryableUnitOfWork
        :IUnitOfWork
{
        IObjectSet<TEntity> CreateSet<TEntity>()
            where TEntity : class;
}
```

Ahora, con este nuevo contrato podríamos definir nuestra clase *Repository* como se ve a continuación:

```
public class Repository<TEntity>
    :IRepository<TEntity>
    where TEntity:class,IObjectWithChangeTracker
{
        #region Members

        IQueryableUnitOfWork _UnitOfWork;

        #endregion

        #region Constructor
```

```
        public Repository(IQueryableUnitOfWork unitOfWork)
        {
            if (unitOfWork == (IQueryableUnitOfWork)null)
                throw new ArgumentNullException("unitOfWork");

            _UnitOfWork = unitOfWork;
        }

        #endregion

        #region IRepository<TEntity> Members

        public IUnitOfWork UnitOfWork
        {
            get { return _UnitOfWork as IUnitOfWork; }
        }

        public void Add(TEntity item)
        {
            if (item == (TEntity)null)
                throw new ArgumentNullException("item");

            //create object set and perform operation
            IObjectSet<TEntity> objectSet =
_UnitOfWork.CreateSet<TEntity>();

            if (item.ChangeTracker != null
                &&
                item.ChangeTracker.State == ObjectState.Added)
            {
                _UnitOfWork.RegisterChanges(item);
            }
        }

        public void Remove(TEntity item)
        {
            if (item == (TEntity)null)
                throw new ArgumentNullException("item");

            //create object set and perform operation
            IObjectSet<TEntity> objectSet =
_UnitOfWork.CreateSet<TEntity>();

            objectSet.Attach(item);

            objectSet.DeleteObject(item);
        }

        public void RegisterItem(TEntity item)
        {
            if (item == (TEntity)null)
                throw new ArgumentNullException("item");

            //create object set and perform operation
            IObjectSet<TEntity> objectSet =
_UnitOfWork.CreateSet<TEntity>();
            objectSet.Attach(item);
        }

        public void Modify(TEntity item)
        {
```

```
            if (item == (TEntity)null)
                throw new ArgumentNullException("item");

            //create object set and perform operation
            if (item.ChangeTracker != null
                &&
                ((item.ChangeTracker.State & ObjectState.Deleted) !=
ObjectState.Deleted)
                )
            {
                item.MarkAsModified();
            }

            _UnitOfWork.RegisterChanges(item);
        }

        public IEnumerable<TEntity> GetAll()
        {
            //create object set and perform operation
            return (_UnitOfWork.CreateSet<TEntity>()).AsEnumerable();
        }

        #endregion
    }
```

Si observamos con más detenimiento alguno de los métodos anteriores, como por ejemplo *Add*, podrá observar como gracias a la llamada al método *CreateSet* ya disponemos de un elemento *IObjectSet<TEntity>* sobre el que podemos interactuar.

Recuerde aquí que *IObjectSet* es el contrato principal de nuestros objetos de consulta de ADO.NET Entity Framework, *ObjectSet* y *ObjectQuery*. Puesto que además se decidió trabajar con *Self Tracking Entitites*, entonces también podemos hacer uso de la información de *ChangeTracker*, con el fin de verificar que el estado de la entidad que intentamos agregar es *Added*. Para finalizar el trabajo de este método *Add*, se hace una llamada al método *RegisterChanges* que, como puede ver en fragmentos anteriores, la implementación de nuestro *IUnitOfWork* en la plantilla del contexto de trabajo generado se corresponde en realidad con la llamada al método extensor *ApplyChanges* que nos proporcionan las plantillas de STE, y que ya hemos explicado en el capítulo anterior de forma extensa.

Lógicamente, si hemos ampliado nuestro contrato de unidad de trabajo en la capa de infraestructura, tendremos que hacer una modificación a la plantilla del contexto de trabajo de *MainModule*, para que, en vez de implementar *IUnitOfWork* implemente ahora *IQueryableUnitOfWork* y de solución al método *CreateSet*. Para ello, solamente tenemos que volver a la plantilla T4, *MainModule.Model.Context.tt*, y volver a modificar la generación de la clase de nuestro contexto, en concreto sustituir *IUnitOfWork* por el nuevo contrato como se puede ver a continuación:

```
<#=Accessibility.ForType(container)#> partial class
<#=code.Escape(container)#> : ObjectContext,IQueryableUnitOfWork
{
```

Fíjese que en este paso tendremos que volver a decirle a nuestra plantilla que conozca un nuevo espacio de nombres, *Infrastructure.Data.Core*. Dejamos en este caso al lector la tarea de incorporarlo tal y como hemos hecho anteriormente.

Por supuesto, como este nuevo contrato define una operación a mayores, tendremos que hacer que se genere en esta plantilla. Para eso bastará con agregar en ésta, antes por ejemplo de nuestro método *Commit*, el siguiente fragmento de código.

```
public IObjectSet<TEntity> CreateSet<TEntity>()
   where TEntity:class
{
            return this.CreateObjectSet<TEntity>();
}
```

Nota: En realidad, entre la colección de pasos que hemos hecho hasta ahora deberíamos incluir uno más. Este paso consiste en separar la definición del código de soporte de nuestras STE a un nuevo ensamblado, de tal forma que, si disponemos de varios módulos en una solución podamos reaprovechar este código. Por simplicidad no se ha hecho en este libro, sin embargo en la aplicación de ejemplo disponible en codeplex puede ver como esta tarea está hecha en *Domain.Core.Entities*.

Llegados hasta aquí, ya tenemos la implementación de nuestra unidad de trabajo terminada, así como nuestra implementación de repositorio genérico. Sin embargo, para terminar nuestra parte de infraestructura aún nos queda por ver la implementación de los repositorios definidos en nuestro dominio, concretamente los contratos *ICustomerRepository* y *IOrderRepository*.

Para ello, partiendo de lo hecho hasta ahora en nuestro ejemplo, incluiremos en nuestro proyecto de infraestructura las implementaciones de estos contratos, **code\ef4_sample4**.

A continuación, en el siguiente fragmento de código podemos ver el cuerpo del repositorio de *Customer* a implementar.

```
public class CustomerRepository
        :Repository<Customer>,ICustomerRepository
{
    #region ICustomerRepository Members

    public IEnumerable<Customer> GetCustomersByFirstName(string
firstName)
    {
       throw new NotImplementedException();
    }

    #endregion
}
```

Como puede observar, este repositorio se apoya, amén de la implementación del contrato *ICustomerRepository*, en la implementación de nuestra clase *Repository* para

dar solución a los distintos métodos comunes de los repositorios. Por lo tanto, nuestra responsabilidad como desarrolladores será la de dar implementación a los métodos específicos de cada contrato. La implementación final del repositorio anterior está expuesta a continuación.

```
public class CustomerRepository
        :Repository<Customer>,ICustomerRepository
{

    #region Constructor

    public CustomerRepository(IQueryableUnitOfWork unitOfWork)
            : base(unitOfWork)
    {
    }

    #endregion

    #region ICustomerRepository Members

    public IEnumerable<Customer> GetCustomersByFirstName(string
firstName)
    {
        return
((IQueryableUnitOfWork)UnitOfWork).CreateSet<Customer>()
                                                    .Where(c =>
c.FirstName.ToLower() == firstName.ToLower())
                                                    .AsEnumerable();
    }

    #endregion
}
```

Para el caso del contrato *IOrderRepository* el trabajo es prácticamente idéntico, exceptuando el método *GetOrderWithDetails*, creado explícitamente para enseñar el caso en el que necesitemos realizar una expansión de consultas.

Como usted sabe del capítulo anterior, ADO.NET Entity Framework, además de los mecanismos de carga perezosa pone también a nuestra disposición la capacidad de realizar expansión de consultas por medio del método *Include*. Este método, está definido para la clase *ObjectQuery<TEntity>*. Sin embargo en el código de nuestros repositorios siempre usamos *IObjectSet<TEntity>*. Para dar solución al uso de este método con este tipo podemos crear un nuevo método extensor, llamado *Include* lógicamente, como el expuesto en el siguiente fragmento.

```
public static IQueryable<TEntity> Include<TEntity>(this
IQueryable<TEntity> queryable, Expression<Func<TEntity, object>>
path)
            where TEntity : class
{
            //analyze expression path

    MemberExpression memberExpression = path.Body as
MemberExpression;
    ObjectQuery<TEntity> query = queryable as ObjectQuery<TEntity>;
    if (memberExpression != null)
```

```
        {
            string memberName = memberExpression.Member.Name;
            if (query != null)
                return query.Include(memberName);
            else
                throw new InvalidOperationException("Invalid queryable");
        }
        else
            throw new InvalidOperationException("Invalid path
    expression!");
            }
```

Fíjese en dos aspectos fundamentales de este método. El primero es que este método extensor está definido sobre *IQueryable<TEntity>*, base para los *IObjectSet<TEntity>*. Además, al contrario que el método *Include* tradicional, en vez de tener un parámetro de cadena de caracteres para indicar la expansión de consultas, ahora pasamos un árbol de expresiones, que nos permitirá indicar la expansión de una forma tipada.

Una vez definido este método extensor, ya podríamos realizar la implementación de nuestro método *GetOrderWithDetails* en el repositorio de *Order* como sigue:

```
public IEnumerable<Order> GetOrderWithDetails(int orderIdentifier)
{
   return ((IQueryableUnitOfWork)UnitOfWork).CreateSet<Order>()
                                    .Include(o=>o.OrderDetails)
                                    .Where(o => o.IdOrder ==
orderIdentifier)
                                    .AsEnumerable();
}
```

2.1.3.- "Testando" ADO.NET Entity Framework 4.1

Una de las críticas más importantes a la primera versión de ADO.NET Entity Framework fue sin duda alguna el insuficiente soporte (y puede que incluso esté siendo generoso con este adjetivo) para realizar pruebas de los distintos componentes de EF que pudiéramos usar en nuestros desarrollos. Estas críticas tuvieron, por suerte para todos, respuestas desde el grupo de producto y con unos pequeños cambios pusieron a nuestra disposición la capacidad de testear y simular nuestras unidades de trabajo y elementos de consulta.

Como se imaginará, a lo largo de este punto no trataremos de enseñarle principios sobre *testing*, *mocking*, etc... y tampoco a usar ningún framework de pruebas. Esto, se escapa completamente al objetivo de este libro, por lo tanto, solamente nos centraremos en mostrar los elementos a nuestra disposición para hacer pruebas y simulaciones. Para ello utilizaremos **MSTest** como framework de pruebas y **Microsoft Pex and Moles** para nuestros mock, stubs (disponible par descarga desde **http://research.microsoft.com/projects/pex/**).

Tal y como hemos hecho durante todos los puntos precedentes, partiremos del trabajo anterior y empezaremos a agregar los distintos elementos que necesitemos a lo largo de este punto, **code\ef4_sample5**. Si revisamos lo que llevamos desarrollado

hasta ahora podríamos resumir que desde el punto de vista de nuestras consultas y trabajo con los repositorios, los elementos fundamentales de uso con EF son los distintos *IObjectSet<TEntity>* usados en la implementación de nuestros repositorios y nuestra unidad de trabajo generada con la plantilla de T4. Por lo tanto, nuestros objetivos para el proyecto de tests serán simular la unidad de trabajo y los elementos de consulta que ésta pueda necesitar usar.

Nota: Revise la definición de la interfaz *IQueryableUnitOfWork* y observe como el método *CreateSet* retorna un tipo *IObjectSet<TEntity>*. Como se podrá imaginar la decisión de devolver este contrato en vez de *ObjectSet<TEntity>* tiene como fundamento posibilitar la creación de simulaciones de estos objetos.

Lo primero que trataremos de hacer será ver cómo intentar simular nuestros objetos de consulta, es decir, los *IObjectSet<TEntity>* que serán usados. Para ello, en nuestro proyecto de tests crearemos una nueva clase que llamaremos *MemorySet*, la cual tiene una estructura como la presentada a continuación en el siguiente fragmento de código.

```
public class MemorySet<TEntity>
       :IObjectSet<TEntity>

{
   #region IObjectSet<TEntity> Members
   public void AddObject(TEntity entity)
   {
      throw new NotImplementedException();
   }

   public void Attach(TEntity entity)
   {
      throw new NotImplementedException();
   }

   public void DeleteObject(TEntity entity)
   {
      throw new NotImplementedException();
   }

   public void Detach(TEntity entity)
   {
      throw new NotImplementedException();
   }

   #endregion

   #region IEnumerable<TEntity> Members

   public IEnumerator<TEntity> GetEnumerator()
   {
      throw new NotImplementedException();
   }
```

```
        #endregion

        #region IEnumerable Members

        System.Collections.IEnumerator
    System.Collections.IEnumerable.GetEnumerator()
        {
            throw new NotImplementedException();
        }

        #endregion

        #region IQueryable Members

        public Type ElementType
        {
            get { throw new NotImplementedException(); }
        }

        public System.Linq.Expressions.Expression Expression
        {
            get { throw new NotImplementedException(); }
        }

        public IQueryProvider Provider
        {
            get { throw new NotImplementedException(); }
        }

        #endregion
    }
```

Como puede observar en el código anterior, la implementación de *IObjectSet* requiere de tres elementos fundamentales. En primer lugar la implementación de un elemento *IQueryable*. Esto es lógico ya que es lo que da soporte a nuestras consultas de LINQ. A mayores, la implementación de una enumeración y, para terminar, distintos métodos como *AddObject* y *DeleteObject*, llevados a esta interfaz precisamente para que podamos testear estas operaciones de forma independiente a nuestro *ObjectContext*, que define también las mismas operaciones.

La implementación de la enumeración, así como de los distintos métodos como *AddObject* y *DeleteObject* es realmente sencilla si apoyamos esta clase por ejemplo en una lista, es decir en un conjunto de elementos en memoria, ya que el método *AddObject* sería tan simple como la inclusión de un nuevo elemento dentro de esta lista y, la enumeración, sería entonces la enumeración de la lista en la que nos estamos apoyando. El caso de las tres propiedades que nos impone *IQueryable* podría ser realmente más complejo, ya que entran en juego elementos muy ligados a los proveedores LINQ. Sin embargo, gracias a un método extensor, *AsQueryable()*, definido en *IEnumerable<TEntity>*, podremos partir también de una lista y dar solución de una forma simple a estas propiedades.

En el siguiente fragmento de código podemos ver la implementación final de muestro *MemorySet*. Para ello nos hemos apoyado en una lista interna inicializada en el constructor de esta clase.

```
public class MemorySet<TEntity>
        :IObjectSet<TEntity> where TEntity:class
{
    List<TEntity> _items;
    public MemorySet(List<TEntity> items)
    {
        if (items == null)
            throw new ArgumentNullException("items");

        _items = items;
    }

    #region IObjectSet<TEntity> Members

    public void AddObject(TEntity entity)
    {
        if (!_items.Contains(entity))
            _items.Add(entity);
    }

    public void Attach(TEntity entity)
    {
        if (!_items.Contains(entity))
            _items.Add(entity);
    }

    public void DeleteObject(TEntity entity)
    {
        if (_items.Contains(entity))
            _items.Remove(entity);
    }

    public void Detach(TEntity entity)
    {
        if (_items.Contains(entity))
            _items.Remove(entity);
    }

    #endregion

    #region IEnumerable<TEntity> Members

    public IEnumerator<TEntity> GetEnumerator()
    {
        return this._items.GetEnumerator();
    }

    #endregion

    #region IEnumerable Members

    System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator()
    {
        return this._items.GetEnumerator();
    }

    #endregion

    #region IQueryable Members
```

```
        public Type ElementType
        {
            get { return _items.AsQueryable().ElementType; }
        }

        public System.Linq.Expressions.Expression Expression
        {
            get { return _items.AsQueryable().Expression; }
        }

        public IQueryProvider Provider
        {
            get { return _items.AsQueryable().Provider; }
        }

        #endregion
    }
```

Ahora, gracias a *MemorySet*, ya tenemos el soporte para simular nuestros objetos de consulta. Por lo tanto pasaremos a realizar nuestra simulación de la unidad de trabajo.

Para ello lo primero en nuestro proyecto de tests será incorporar el archivo de *Moles* (figura 5.10) del ensamblado en el que se define nuestro contrato de unidad de trabajo, *IQueryableUnitOfWork*, *Infrastructure.Data.Core*.

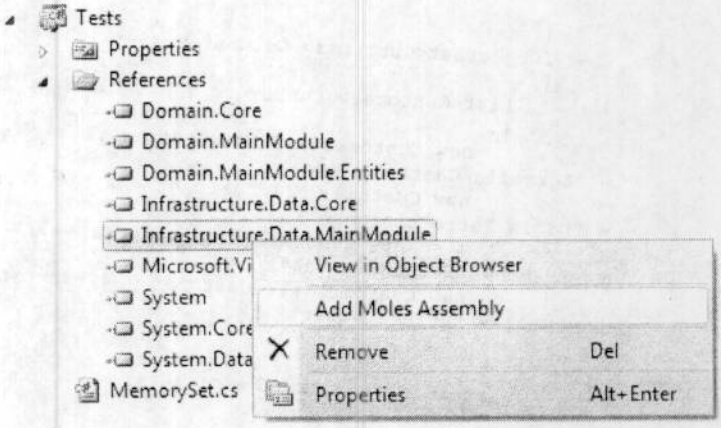


Figura 5.10.- Añadir el ensamblado de Moles

Una vez agregado el archivo de Moles generalmente es necesario compilar. Una vez incluido en el proyecto ya podremos establecer nuestro *stub* de la unidad de trabajo. A continuación, se muestra una configuración del *stub* de nuestro contrato *IQueryableUnitOfWork* por medio de la clase *StubMainModuleUnitOfWork*. En este *stub* solamente se ha configurado el *IObjectSet* para nuestra entidad *Customer*.

Lógicamente para el resto de elementos se hará de una forma similar, dejamos al lector esto como tarea

Nota: Por supuesto usted es libre de usar cualquier framework de *mocks* y *stubs*, no tiene por qué usar este. Hay muchos y variados, tanto de pago como gratuitos. A continuación puede ver algunos de los más conocidos:

- NUnit.Mocks, **http://nunit.org/**
- NMock, **http://nmock.org/nmock1-documentation.html**
- RhinoMocks, **http://www.ayende.com/projects/rhino-mocks.aspx**
- TypeMock, **http://www.typemock.com/**

```
public class StubMainModuleUnitOfWork
      :Infrastructure.Data.Core.Moles.SIQueryableUnitOfWork
{
    public StubMainModuleUnitOfWork()
    {
        InitializeSets();
    }

    private void InitializeSets()
    {
        this.CreateSet<Customer>(
                                  ()
                                  =>
                                  {
                                     return GetCustomerSet();
                                  });

    }
    IObjectSet<Customer> GetCustomerSet()
    {
        List<Customer> customers = new List<Customer>()
        {
            new Customer(){IdCustomer = 1,FirstName = "Unai",LastName
= "Zorrilla Castro"},
            new Customer(){IdCustomer = 2,FirstName = "Cesar",LastName
= "De la Torre Llorente"},
                  new Customer(){IdCustomer = 3,FirstName = "Yamil",
LastName = "Hernandez Saa"},
            new Customer(){IdCustomer = 4,FirstName = "Pablo",
LastName = "Pelaez Aller"}
        };

        return customers.ToMemorySet();
    }
}
```

Las pruebas de nuestros repositorios podrán ahora realizarse sin necesidad de utilizar el contexto generado por EF, como podemos ver en el siguiente fragmento del código de las pruebas.

```
[TestClass()]
public class CustomerRepositoryTests
{
    [TestMethod()]
    public void Mock_GetCustomerByFirstName_Invoke_Tests()
    {
        //Arrange
        IQueryableUnitOfWork unitOfWork = new
StubMainModuleUnitOfWork();

        ICustomerRepository customerRepository = new
CustomerRepository(unitOfWork);

        //Act
        Customer customer =
customerRepository.GetCustomersByFirstName("Unai")
                  .SingleOrDefault();

        //Assert
        Assert.IsNotNull(customer);
    }
}
```

Lógicamente, lo que estamos presentando aquí son ejemplos simples, no los tome como norma para manejar sus simulaciones ni la forma de realizar las pruebas. El objetivo es presentarle las opciones técnicas que tiene a su disposición para esta tarea.

Si quiere profundizar más sobre cómo realizar las pruebas de capas de persistencia, manejo de las unidades de trabajo y otros elementos importantes, le recomendamos que revise la aplicación de ejemplo mencionada al comienzo de este capítulo.

2.2.- STE y las entidades duplicadas

En principio este punto no estaba pensado para ser incluido en el libro. Sin embargo después de distintas revisiones de los problemas con los que se encuentra la comunidad, se decidió intentar aportar un poco de luz en el problema más recurrente de todos que, como el título de este punto indica, es el de la **duplicación de entidades**.

Para intentar mostrar el problema, y por supuesto, también los distintos *workarounds* partiremos del siguiente modelo de tablas, expuesto en el siguiente fragmento SQL.

```
--SQL\EF4_EX_16.sql

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Customer](
    [IdCustomer] [int] IDENTITY(1,1) NOT NULL,
    [FirstName] [nvarchar](max) NOT NULL,
    [LastName] [nvarchar](max) NOT NULL,
 CONSTRAINT [PK_Customer] PRIMARY KEY CLUSTERED
(
```

```
    [IdCustomer] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Countries](
    [IdCountry] [int] IDENTITY(1,1) NOT NULL,
    [Name] [nvarchar](max) NOT NULL,
 CONSTRAINT [PK_Countries] PRIMARY KEY CLUSTERED
(
    [IdCountry] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Orders](
    [IdOrder] [int] IDENTITY(1,1) NOT NULL,
    [Total] [decimal](18, 0) NOT NULL,
    [IdCustomer] [int] NOT NULL,
    [IdCountry] [int] NOT NULL,
 CONSTRAINT [PK_Orders] PRIMARY KEY CLUSTERED
(
    [IdOrder] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
GO
CREATE NONCLUSTERED INDEX [IX_FK_CountryOrder] ON [dbo].[Orders]
(
    [IdCountry] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY = OFF, DROP_EXISTING = OFF,
ONLINE = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON
[PRIMARY]
GO
CREATE NONCLUSTERED INDEX [IX_FK_CustomerOrder] ON [dbo].[Orders]
(
    [IdCustomer] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY = OFF, DROP_EXISTING = OFF,
ONLINE = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON
[PRIMARY]
GO

ALTER TABLE [dbo].[Orders]  WITH CHECK ADD  CONSTRAINT
[FK_CountryOrder] FOREIGN KEY([IdCountry])
REFERENCES [dbo].[Countries] ([IdCountry])
GO
```

```
ALTER TABLE [dbo].[Orders] CHECK CONSTRAINT [FK_CountryOrder]
GO

ALTER TABLE [dbo].[Orders]  WITH CHECK ADD  CONSTRAINT
[FK_CustomerOrder] FOREIGN KEY([IdCustomer])
REFERENCES [dbo].[Customer] ([IdCustomer])
GO
ALTER TABLE [dbo].[Orders] CHECK CONSTRAINT [FK_CustomerOrder]
GO
```

Sobre esta estructura relacional crearemos un proyecto, **ef4_ex_40**, con un modelo de EDM (figura 5.11) en el que estableceremos el artefacto de generación STE. Una vez hecho esto, intentaremos poner distintos ejemplos para revisar el problema que nos ocupa.

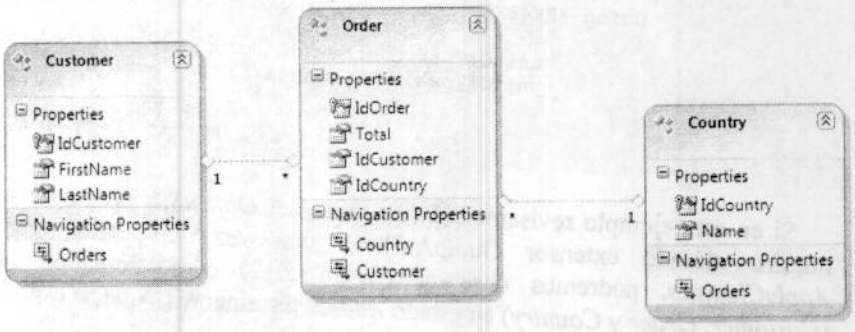


Figura 5.21.- Model de Entity Data Model

Lo primero que haremos será ver un ejemplo de uso de estas entidades, para simular el trabajo desconectado con STE. Simplemente haremos el proceso de serialización y deserialización de la misma forma que lo haría un servicio de WCF, por ejemplo.

En el primer caso, a continuación, podemos ver como se inserta en el modelo anterior un nuevo cliente, con un pedido y un país asociado, elementos con los que trabajaremos posteriormente.

```
class Program
{
    static void Main(string[] args)
    {
        Customer customer = new Customer()
        {
            FirstName = "Maria",
            LastName = "Aira"
        };

        Order newOrder = new Order()
        {
            Total = 100M,
            Country = new Country(){Name = "Spain"}
```

```
        };

        customer.Orders.Add(newOrder);

        SaveCustomer(WCF(customer));
    }


    static TEntity WCF<TEntity>(TEntity item)
            where TEntity:class
    {
        DataContractSerializer serializer = new
DataContractSerializer(typeof(TEntity));

        MemoryStream stream = new MemoryStream();

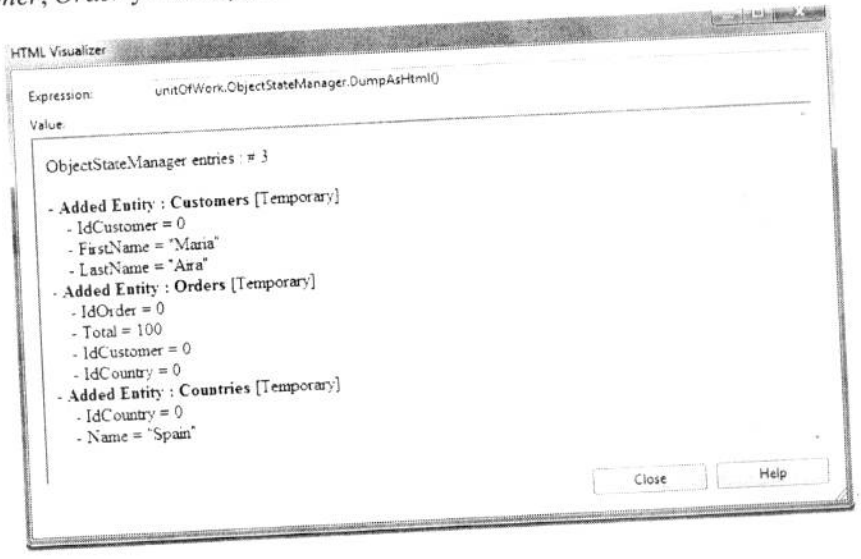
        serializer.WriteObject(stream, item);
        stream.Seek(0, SeekOrigin.Begin);

        return (TEntity)serializer.ReadObject(stream);
    }

    static void SaveCustomer(Customer customer)
    {
        using (EF4EXEntities unitOfWork = new EF4EXEntities())
        {
                unitOfWork.Customers.ApplyChanges(customer);
                unitOfWork.SaveChanges();

        }
    }
}
```

Si en este ejemplo revisamos el contenido del *ObjectStateManager* asociado con nuestro método extensor DumpAsHtml, una vez se realiza una llamada a *ApplyChanges*, podremos observar (figura 5.12) como tenemos tres entidades (*Customer*, *Order* y *Country*) en estado *Added*, justamente lo que se esperaría.



```
        {
            using (EF4EXEntities unitOfWork = new EF4EXEntities())
            {
                unitOfWork.Customers.ApplyChanges(customer);
                unitOfWork.SaveChanges();
            }
        }
    }
```

Observe como en el código anterior cuando se recupera el cliente para un determinado identificador, junto con este elemento también se incluye la información de sus pedidos y los países asociados a ellos. Ahora si en el cliente agregamos un nuevo pedido, como es nuestro caso, y en este pedido establecemos el valor de la propiedad *Country*, podríamos romper la consistencia del grafo. ¿Cómo es posible esto?. Dese cuenta que el pedido obtenido para nuestro cliente ya incluye el país al que pertenece, en este caso también con identificador 2. Si en el nuevo pedido establecemos la propiedad *Country* con una nueva instancia de este mismo identificador, entonces, en el grafo tendríamos dos referencias distintas que representan en realidad al mismo elemento. Al llevar estos cambios a nuestro contexto y llamar al método *ApplyChanges*, el API de STE se encontrará con dos entidades con la misma clave, es decir, dos entidades duplicadas y se lanzará una excepción con el siguiente mensaje.

```
AcceptChanges cannot continue because the object's key values conflict
with another object in the ObjectStateManager. Make sure that the key
values are unique before calling AcceptChanges.
```

Aunque nosotros hemos "forzado" en exceso el ejemplo para intentar reproducir este problema, en realidad, hay escenarios dónde este comportamiento podría llegar a ocurrir con determinada frecuencia. Lógicamente, para estos casos, disponemos de distintos *workaround* que podríamos aplicar.

El primero, y también el más sencillo consiste en utilizar las "foreign key properties" siempre que estemos asignando una propiedad de navegación. En nuestro caso, bastaría con cambiar el proceso de creación de un nuevo pedido tal y como se ve a continuación:

```
Customer customerInClient = WCF(GetCustomerWithOrder(2));
Country country = WCF(GetCountry(2));

Order order = new Order()
{
   Total = 123M,
   IdCountry = 2
};

customerInClient.Orders.Add(order);

SaveCustomer(WCF(customerInClient));
```

Si revisamos, al igual que hicimos anteriormente, el contenido de nuestro *ObjectStateManager* contendrá la información correcta y sin duplicados, figura 5.13.

Figura 5.3.- Contenido del ObjectStateManager

Si modificamos este código para, en vez de agregar un cliente, un pedido y un país agregar solamente un nuevo pedido para un cliente existente, se nos podría llegar a producir el problema que nos ocupa. Fíjese en el siguiente fragmento:

```
class Program
{
    static void Main(string[] args)
    {
        Customer customerInClient = WCF(GetCustomerWithOrder(2));

        Country country = WCF(GetCountry(2));

        Order order = new Order()
        {
            Total = 123M,
            Country = country
        };

        customerInClient.Orders.Add(order);

        SaveCustomer(WCF(customerInClient));
    }

    static TEntity WCF<TEntity>(TEntity item)
          where TEntity:class
    {
        DataContractSerializer serializer = new
DataContractSerializer(typeof(TEntity));

        MemoryStream stream = new MemoryStream();

        serializer.WriteObject(stream, item);
        stream.Seek(0, SeekOrigin.Begin);

        return (TEntity)serializer.ReadObject(stream);
    }
    static Customer GetCustomerWithOrder(int idCustomer)
    {
        using (EF4EXEntities unitOfWork = new EF4EXEntities())
        {
            return unitOfWork.Customers
                             .Include("Orders.Country")
                             .Where(c => c.IdCustomer == idCustomer)
                             .SingleOrDefault();
        }
    }
    static Country GetCountry(int idCountry)
    {
        using (EF4EXEntities unitOfWork = new EF4EXEntities())
        {
            return unitOfWork.Countries
                             .Where(c => c.IdCountry == idCountry)
                             .SingleOrDefault();
        }
    }

    static void SaveCustomer(Customer customer)
```

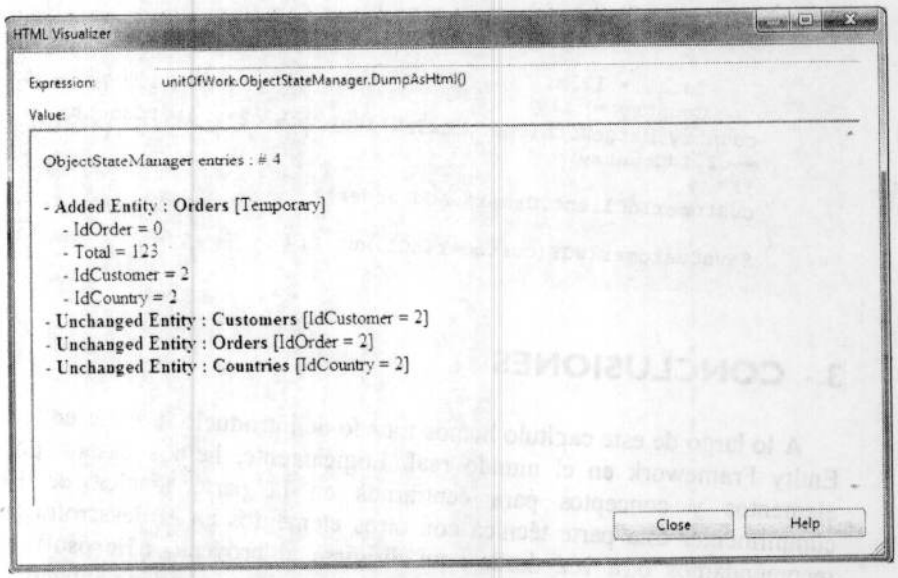


Figura 5.4.- Contenido del objectStateManager

Otra posibilidad, a mayores de la anterior, es asegurarnos que cuando se establece una propiedad de navegación, en el grafo del objeto no hay presente ninguna entidad que represente al mismo elemento. Para asegurarnos esto, podríamos servirnos de algún método de ayuda que realice esta tarea.

Un ejemplo de un método de ayuda podría ser el siguiente:

```
public static TEntity MergeWith<TEntity, TGraph>(this TEntity entity,
TGraph graph, Func<TEntity, TEntity, bool> keyComparer)
            where TEntity : class,IObjectWithChangeTracker
            where TGraph : class,IObjectWithChangeTracker
{
   using (ChangeTrackerIterator iterator =
ChangeTrackerIterator.Create(graph))
   {
      return iterator.OfType<TEntity>().SingleOrDefault(e =>
keyComparer(entity, e)) ?? entity;
   }
}
```

Nota: En este método *MergeWith*, se utiliza un iterador de entidades capaz de moverse a través de un grafo. Con el fin de no incluir código en este libro que pueda confundir al lector si le interesa lo puede revisar directamente en el ejemplo code\ef4_ex_40.

Ahora, con este método de ayuda, *MergeWith*, podríamos rehacer nuestro código de ejemplo como sigue, y los resultados volverían a ser correctos:

```
Customer customerInClient = WCF(GetCustomerWithOrder(2));
Country country = WCF(GetCountry(2));

Order order = new Order()
{
    Total = 123M,
    Country =
country.MergeWith(customerInClient,(c1,c2)=>c1.IdCountry
==c2.IdCountry)
};
customerInClient.Orders.Add(order);

SaveCustomer(WCF(customerInClient));
```

3.- CONCLUSIONES

A lo largo de este capítulo hemos tratado de introducir al lector en la utilización de Entity Framework en el mundo real. Lógicamente, hemos pasado por alto ciertos elementos y conceptos para centrarnos en la parte técnica de EF. Si desea cumplimentar esta parte técnica con otros elementos en el desarrollo de software le recomendamos otra vez desde aquí dirigirse al proyecto **Microsoft NLayer App**, **http://microsoftnlayerapp.codeplex.com**, y revisar la documentación del sitio de arquitectura de MSDN en castellano, **http://msdn.microsoft.com/es-es/architecture**.

CAPÍTULO

Entity Framework 4.1

1.- INTRODUCCIÓN

Mientras se escribía este libro en versión impresa, el equipo de desarrollo de Data Programability de Microsoft en Redmond, estaba ultimando un "pequeño" agregado a esta segunda versión de ADO.NET Entity Framework. Puesto que este agregado en el instante de la impresión estaba relativamente poco maduro, se tomó la decisión -en conjunto con el editor- de no incluir por defecto nada relativo a este tema, temiendo (como así ha sido por otra parte) que los cambios en la versión final fueran abundantes con respecto al material del que se disponía en ese momento.

Nota: EF 4.1 se ha construido sobre la base de EF 4.1, por lo tanto no es una versión nueva sino un aditivo incorporado a las bases de las que disponemos, en forma de una nueva librería llamada EntityFramework.dll. Hasta donde se sabe y se puede contar, este nuevo agregado, versión 4.1, no será el único que seguirá saliendo del edificio 33 del campus de Redmond. La idea, será evolucionar el producto con ciclos más cortos que .NET Framework o Visual Studio, algo a lo que estábamos acostumbrados hasta ahora y que parece una excelente noticia

Aunque en el párrafo anterior hemos aplicado el adjetivo "pequeño", en realidad las nuevas posiblidades incluidas en esta extensión son realmente impresionantes. Más alla de nuevas capacidades de modelado, también se nos abren nuevas estrategias de división del trabajo en modelos orientados al dominio, así como formas de abordar los proyectos. A lo largo de este capítulo, iremos abordando todas las novedades de una forma ordenada, intentando aprovechar los conocimientos adquiridos después de la lectura del libro.

2.- INSTALACIÓN Y PUESTA EN MARCHA

Lo primero es lo primero, y, por lo tanto, empezaremos por revisar el proceso de instalación de EF 4.1. Acertadamente para la mayoría de los que hemos tenido el placer de participar en las versiones preliminares, el equipo de ADO.NET pensó que una buena forma de distribuir las nuevas funcionalidades de EF era mediante un paquete de NuGet.

Nota: NuGet es un proyecto Open Source alojado en CodePlex, **http://nuget.codeplex.com/**, que se encarga de administrar y gestionar "paquetes" de bibliotecas de terceros. Si no lo conoce le recomiendo que no pierda más tiempo y agrege la extensión de NuGet a su Visual Studio directamente desde el "Extension Manager"

Una vez que disponemos de NuGet en Visual Studio solamente tenemos que incluir un paquete llamado "Entity Framework", bien desde la consola de comandos de NuGet, en la siguiente figura, bien desde la nueva opción presente en el menú contextual de las referencias de un proyecto .NET:

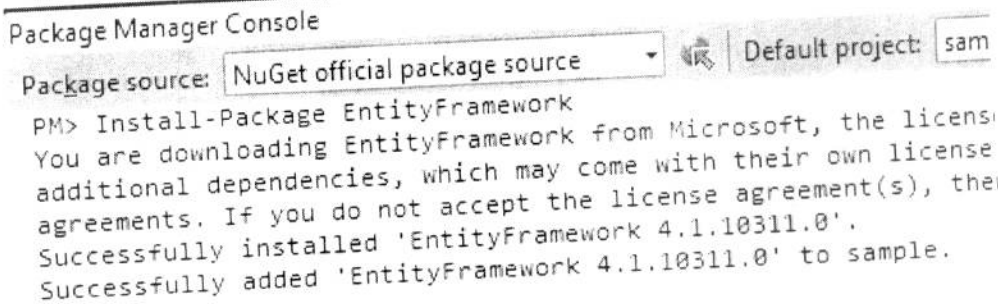


Figura 1.- Agregando el paquete de EF 4.1

Una vez agregado el paquete "Entity Framework" ya tendremos disponible una nueva referencia EntityFramework.dll con las novedades contenidas en esta nueva versión, Figura 2.

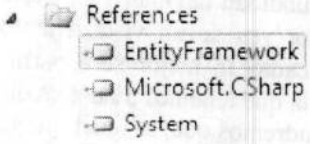


Figura 2.- Biblioteca de ADO.NET EF 4.1

3.- UN NUEVO MODELO DE TRABAJO

Hasta ahora si queríamos trabajar con ADO.NET Entity Framework 4.1 teníamos distintas opciones a la hora de crear nuestro modelo de entidades. La primera, partiendo de cero con nuestro **Entity Data Model** diseñando el dominio, más comúnmente conocida como **Model First**. Lógicamente, también podemos partir de un modelo creado a partir de la definición de un determinado esquema relacional (de hecho es lo más usado hoy por hoy) el cual, lógicamente, podremos ir adaptando posteriormente. Esta opción se suele conocer como **Database First**.

En estos dos escenarios, o formas de abordar el diseño del dominio, el denominador común es **EDM** como pieza fundamental en la que basarse para la creación del modelo. Si bien **Entity Data Model** nos ofrece una buena herramienta de trabajo, también implica una serie de restricciones que podrían hacer no necesario/recomendable este elemento. El trabajo con modelos de grandes dimensiones, los problemas de *tooling* dentro de Visual Studio o los refrescos de los modelos de dominio ante cambios frecuentes son parte de los problemas/inconvenientes que podríamos encontrarnos. Con el fin de relajar estos inconvenientes, o bien por una mayor sintonía con la forma de trabajo que iremos presentando a lo largo de este capítulo, el equipo de ADO.NET nos ofrece la posibilidad de diseñar nuestro modelo de entidades haciendo uso solamente de código, sin necesidad de utilizar un **edmx** para la definición y el mapeo de nuestras entidades. Esta es sólo otra de las características incluidas en esta nueva versión.

Nota: Algunos de los problemas de rendimiento en el *Tooling* de Visual Studio han sido relajados con el Service Pack 1 de Visual Studio 2010.

3.1.- Un primer ejemplo, los elementos fundamentales

Los elementos fundamentales para que esta nueva forma de trabajo (**Code First** y todo lo que conlleva) puedan estar a nuestra disposición, se podrían resumir en los siguientes puntos, mostrados a continuación:

- Conexiones y modelado
- Convenciones
- Nuevas API'S

Lógicamente, cada uno de estos elementos los iremos desglosando poco a poco, deteniendonos, con la profundidad habitual ya en este libro, en cada uno de ellos. Sin embargo, para que el lector vea de un vistazo tanto la forma de trabajar como los cambios en las áreas indicadas lo mejor será partir de un ejemplo simple que nos muestre parte de la potencia que tenemos a nuestra disposición.

Para este ejemplo supondremos que, al igual que hacíamos de forma tradicional con ADO. NET EF 4.1, queremos crear un modelo de entidades. La principal diferencia es que ahora directamente nos pondremos manos a la obra diseñando como queremos que sean estas entidades, sin necesidad de pasar por una herramienta de modelado como EDM.

```
public class Customer
{
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public ICollection<Order> Orders { get; set; }
}

public class Order
{
    public int OrderId { get; set; }
    public decimal TotalOrder { get; set; }
    public int CustomerId { get; set; }
    public Customer Customer { get; set; }
}
```

Con estas sencillas clases, y sin necesidad de especificar ninguno de los tradicionales espacios de nuestros ficheros edmx (**CSDL**, **SSDL** y **MSL**) ya tenemos definido nuestro modelo de entidades.

Nota: Por supuesto, tal y como acaba de ver, las clases a utilizar dentro de Code First son clases POCO, elemento que cuya definición ya conoce del capítulo de este libro dedicado a las distintas opciones de generación de entidades. Puesto que es habitual que haya cierta confusión, viene al caso recordar aquí que el concepto de POCO surge por el hecho de intentar mantener el principio de "Persistent Ignorant" o Ignorancia de la Persistencia. Es decir, que las entidades de nuestro dominio no tengan por qué conocer ningún elemento base o contrato sobre la tecnología usada para persistirlas.

Una vez que ya tenemos definido el modelo, el siguiente paso será el de crear nuestra unidad de trabajo, al igual que antes, no necesitamos ningún componente de *tooling* para realizarlo, podemos ir directamente al código:

```
public class CustomerContext
        :DbContext
{
   public IDbSet<Customer> Customers {get;set;}
   public IDbSet<Order> Orders {get;set;}
}
```

Así, sin más preámbulos, ya estaríamos dispuestos para trabajar con nuestras dos entidades *Customer* y *Order*, haciendo nuestras operaciones de CRUD.

```
using (CustomerContext context = new CustomerContext())
{
   //El primer cliente y luego los pedidos
   Customer customer = context.Customers.First();

   foreach (Order order in customer.Orders)
   {
       //TODO: hacer algo con los pedidos
   }
}
```

Recuerde como, todo esto, lo hemos hecho **sin crear la base de datos, sin especificar nuestra cadena de conexión**, ni de **ningún elemento extra al código** que estamos viendo. Lógicamente, aquí no hay magia, sobre los elementos que permiten que todo esto funcione de una forma tan transparente por ahora y cómo podríamos modificar sus comportamientos por defecto es algo que iremos viendo, como comentábamos al principio de este punto, a lo largo de este mismo capítulo.

3.2.- Las conexiones y el modelo de datos

En el ejemplo que acabamos de mostrar en el punto 2.1 todo ha funcionado sin mayores problemas, si hiciéramos pruebas insertando, borrando o realizando cualquier operación de selección o mantenimiento todo funcionaría de una forma correcta. El hecho de no haber necesitado especificar ninguna cadena de conexión es, porque por defecto, **EF Code First** presupone una cadena de conexión, basada en **Sql Server Express** y un nombre de catálogo igual al nombre perfectamente cualificado de nuestra clase contexto. Es decir, la conexión que se utilizaría por defecto para nuestro ejemplo anterior sería la siguiente:

```
"Server=.\SQLEXPRESS;Initial
Catalog=[Namespace].CustomerContext;Integrated Security=true"
```

Nota: En muchas ocasiones nos referiremos a EF Code First incluyendo aquí el uso de los nuevos elementos del API como *DbContext*. En realidad, esto es una imprecisión puesto que *DbContext* podría funcionar también perfectamente con un modelo EDM. Más adelante intentaremos establecer claramente qué es cada cosa.

Este comportamiento por defecto, viene definido en una nueva clase llamada **Database**, clase que está disponible dentro del espacio de nombres **System.Data.Entity**. Esta clase, además de los diferentes métodos de instancia que nos ofrece, y que veremos más adelante, dispone de una propiedad estática llamada **DefaultConnectionFactory**.

```
public class Database
{
public static IDbConnectionFactory DefaultConnectionFactory { get;
set; }
// Omitido por brevedad
}
```

DefaultConnectionFactory nos permitirá especificar nuestra factoría de creación de conexiones a partir de un nombre o una cadena de conexión.

```
public interface IDbConnectionFactory
{
        DbConnection CreateConnection(string nameOrConnectionString);
}
```

Nota: Fíjese como estos elementos utilizan los elementos definidos en **System.Data.Common** para dar una jerarquía común a todos los proveedores de ADO.NET.

Por defecto también dentro del mismo espacio de nombres anterior, *System.Data.Entity.Infrastructure*, dispondremos de dos factorías de conexión: **SqlConnectionFactory** y **SqlCeConnectionFactory**. Cómo se habrá imaginado la primera hace referencia a un factoría para las conexiones contra la familia de **Sql Server** y la segunda para las conexiones con **Sql Compact Edition**. Si no especificamos ninguna de estas dos, usando para ello la propiedad **DefaultConnectionFactory** vista hace unos párrafos, tendremos asignada por defecto la factoría para las conexiones contra Sql Server. Como hemos visto esta factoría define por defecto una cadena de conexión base contra Sql Server apuntando a una instancia definida como .\SQLEXPRESS y con seguridad integrada. Esto mismo, lo podríamos comprobar si viéramos el código de nuestra clase en nuestra herramienta de desensamblado favorita:

```
// Methods
public SqlConnectionFactory()
{
  this._baseConnectionString = @"Data Source=.\SQLEXPRESS; Integrated Security=True; MultipleActiveResultSets=True";
}
```

Figura 3.- Constructor de SqlConnectionFactory

Una vez que tenemos la cadena base, como ya dejamos entrever anteriormente, solamente nos quedaría indicar el catálogo al que nos queremos conectar. Para ello el contrato **IDbConnectionFactory** en su método **CreateConnection** nos permite establecer este catálogo, en nuestro ejemplo, con el uso de **SqlConnectionFactory**. Como este nombre no se ha establecido, el valor que se asigna por defecto es el path completamente cualificado de la clase contexto, algo similar a:

EntityFramework.Samples.CustomerContext

3.2.1.- Modificación de los parámetros de conexión

Lógicamente, la mayoría de las veces necesitaremos adaptar nuestras cadenas de conexión, bien porque queramos agregar nuevas especificaciones o bien porque queramos cambiar los parámetros de conexión, servidor, seguridad etc. Para esta tarea disponemos de distintas alternativas en función de lo que queramos realizar. Lo primero que veremos es cómo modificar el nombre del catálogo al que queremos conectarnos. Para ello simplemente necesitamos hacer uso de uno de los posibles constructores de nuestra unidad de trabajo.

El siguiente código es un ejemplo de esto:

```
public class CustomerContext
        :DbContext
{
   public CustomerContext()
    :base("CRMDB")
  {
  }

  public IDbSet<Customer> Customers { get; set; }
  public IDbSet<Order> Orders { get; set; }
}
```

En realidad, si revisamos el código del constructor de DbContext, veríamos como el parámetros que hemos establecido podría ser de forma indistinta el nombre del catálogo de la base de datos o también la cadena de conexión completa que queremos usar, de tal forma, que por ejemplo si quisiéramos conectarnos con la siguiente cadena de conexión *Server=.;Initial Catalog=CRMDB;User Id=user;Password=password* solamente tendríamos que escribir lo siguiente:

```
public class CustomerContext
```

```
  :DbContext
{
public CustomerContext()
  :base("Server=.;Initial Catalog=CRMDB;Use=user;Password=password")
{
}

public IDbSet<Customer> Customers { get; set; }
  public IDbSet<Order> Orders { get; set; }
}
```

Seguramente, llegados a este punto se estará preguntando si estos mismos parámetros pueden ser establecidos y modificados usando los ya conocidos archivos de configuración, más concretamente la sección dedicada a las cadenas de conexión, habitual tanto en EF como en otros elementos. La única condición en este caso es establecer como nombre de la cadena de conexión el *path* completamente cualificado del contexto de trabajo:

```
<?xml version="1.0"?>
<configuration>
  <connectionStrings>
    <add name="[Namespace].CustomerContext"
connectionString="Server=.\SQLEXPRESS;Initial
Catalog=SampleCustomer;Integrated Security=true"
providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

3.2.2.- Extendiendo nuestros IDbConnectionFactory

En el punto anterior hemos visto como, a partir de una factoria de conexiones por defecto, podíamos cambiar el nombre o los parámetros de la cadena de conexión. Otras veces, en ocasiones, nos interesa no solamente poder cambiar los parámetros que deseamos por omisión sino también cambiar la base de datos de destinto. Para ello, como ya hemos comentado, tenemos nuestro contrato **IDbConnectionFactory** y las implementaciones por defecto, **SqlConnectionFactory** y **SqlCeConnectionFactory**, factoria para Sql Compact Edition. Como ejemplo de extensibilidad veremos como crear nuestra propia factoria para Sql Server, donde en lugar de utilizar Sql Express como instancia por defecto, se utilice la instancia local.

```
public class LocalSqlConnectionFactory
      :IDbConnectionFactory
    {
      string baseConnectionString = string.Empty;

      public LocalSqlConnectionFactory()
      {
          baseConnectionString = "Server=.;Integrated
Security=true;MultipleActiveResultSets=true";
      }
      public System.Data.Common.DbConnection
```

```
CreateConnection(string catalogName)
        {
            SqlConnectionStringBuilder builder = new
SqlConnectionStringBuilder(baseConnecti
onString);
            builder.InitialCatalog = catalogName;

            return new SqlConnection(builder.ConnectionString);
        }
    }
```

Una vez que lo hemos creado, para establecerlo, solamente tenemos que setearlo en la clase DbDatabase como se ve a continuación:

```
DbDatabase.DefaultConnectionFactory = new
LocalSqlConnectionFactory();
```

3.2.3.- Inicializadores de conexión

A lo largo de este punto hemos visto como utilizando los valores por defecto o con cualquier uso de nuestros **IDbConnectionFactory** se podía obtener la cadena de conexión a la base de datos con la que se va a trabajar. Sin embargo en ningún caso hemos visto cómo hacer que esa base de datos para nuestro código se genere de forma automática como pasaba para nuestro ejemplo inicial. Como iremos viendo, esto se realiza por medio de lo que se conoce como inicializadores. Estos incializadores vienen definidos por el contrato *IDatabaseInitializer<TContext>* el cual se define como se ve a continuación:

```
public interface IDatabaseInitializer<in TContext>
where TContext : global::System.Data.Entity.DbContext
{
  void InitializeDatabase(TContext context);
}
```

Como se puede observar, el método **InitializeDatabase** nos permite especificar las acciones que queramos que se ejecuten antes de realizar ninguna operación con la misma. Por defecto, el grupo de producto nos ofrece dos inicializadores, llamados **DropCreateDatabaseAlways** y **DropCreateDatabaseIfModelChanges.** Como habrá deducido por sus nombres autoexplicativos, el primero nos recrea automáticamente la base de datos cada vez que se va a usar, en el ámbito del domino de aplicación. Es decir, si hay dos llamadas al uso de una unidad de trabajo, solamente se recrearía la base de datos en la primera llamada. Por el contrario, con el valor enumerado **DropCreateDatabaseIfModelChanges** nos recrearía la base de datos solamente si el modelo de nuestro código no se correspondiera con la misma. Dentro de poco veremos como es posible verificar esto.

Si revisa el código de los inicializadores que tenemos por defecto, podrá observar como ámbos nos ofrecen un método virtual llamado **Seed**, gracias al cual además de

realizar el trabajo de recreación de la base podríamos implementar algún elemento extra, como por ejemplo una inicialización de datos de ejemplo.

Nota: Otro posible uso de estos *inicializadores sobre el que no nos* vamos a extender es el proceso de migraciones de proyectos de unos esquemas anteriores a una serie de nuevos esquemas. Sobre este tema, el grupo de producto de ADO.NET está trabajando fuerte, para que, seguramente, en la siguiente versión del producto podamos tener algo disponible con respecto a este tema. Le recomiendo desde aquí que la revisión de la siguiente entrada de EF DESIGN, blog del equipo de ADO.NET dedicado a mostrar elementos de diseño en los que están trabajando.

http://blogs.msdn.com/b/efdesign/archive/2010/10/22/code-first-database-evolution-aka-migrations.aspx

Para demostrar el uso del método *Seed*, partiremos del inicializador **DropCreateDatabaseAlways** y haremos una subclase de ésta con una inicialización de datos por defecto.

```
public class SeedWithDataInitializer
        : DropCreateDatabaseAlways<CustomerContext>
{
    protected override void Seed(CustomerContext context)
    {
        List<Customer> customers = new List<Customer>()
        {
            new Customer(){CustomerId = 1,FirstName =
"Unai",LastName="Zorrilla Castro"},
            new Customer(){CustomerId = 2,FirstName = "Lucia",
LastName = "Zorrilla Aira"}
        };

        //add items
        customers.ForEach(c => context.Customers.Add(c));

        //complete unit of work
        context.SaveChanges();
    }
}
```

Una vez asignado este inicializador si revisaramos los elementos de tipo *Customer* en una consulta, podríamos ver los datos por defecto incluidos en el código anterior, como era de esperar por otra parte.

```
DbDatabase.SetInitializer<CustomerContext>(new
SeedWithDataInitializer());
```

3.3.- Convenciones y mapeos

En el punto anterior, 2.2, hemos podido ver cuáles son los mecanismos por los cuales EF Code First asigna y establece la cadena de conexión para la base de datos a utilizar, gracias a lo cual, como vimos en nuestro primer ejemplo, sin necesidad de realizar ninguna configuración/parametrización podemos hacer operaciones de consulta y mantenimiento. Sin embargo, como se ha obtenido la definición del esquema de la base de datos es un "misterio" que aún no sabemos, pero que trataremos de resolver en las siguientes líneas. El como a partir de las definiciones de nuestras entidades *Customer* y *Order* se ha generado el esquema correspondiente, se basa en el concepto de convenciones, es decir, en *"normas o prácticas admitidas tácitamente"*. Estas convenciones son tan variadas como las que asignan como nombre de la tabla para una entidad el nombre de la propia entidad o bien las que asignan como clave primaria de una tabla a aquellas propiedades de una entidad que se llamen **Id** o bien el nombre de la entidad con el postfijo **Id**, como es nuestro caso con *CustomerId*.

Actualmente existen muchas convenciones que EF Code First pone a nuestra disposición, aunque como es lógico podremos eliminar aquellas que no nos interesen en nuestros proyectos.

Nota: Hasta la CTP 5 de Code First el sistema de convenciones nos ofrecía la posibilidad de crear nuestras propias convenciones y agregar las mismas al motor de Code First, característica conocida como "Plugable Conventions". Lamentablemente, en la versión final esta característica no está presente puesto que no se llegó a los estándares de calidad que exigía el grupo de producto. Lógicamente es de suponer que esta característica estará en futuras versiones.

3.4.- Convenciones por defecto

Aunque en realidad no existen otro tipo de convenciones más que las que EF 4.1 dispone, nos gustaría otorgarles este adjetivo puesto sabemos que en un futuro próximo esto no será así, es decir, podremos llegar a crear nuevas convenciones, pero para esto habrá que esperar.

La lista de convenciones por defecto puede verse a continuación en la siguiente tabla:

Tabla 7.- Lista de convenciones por defecto

Convención
NotMappedTypeAttributeConvention
ComplexTypeAttributeConvention
TableAttributeConvention
NotMappedPropertyAttributeConvention

KeyAttributeConvention
RequiredPrimitivePropertyAttributeConvention
RequiredNavigationPropertyAttributeConvention
TimestampAttributeConvention
ConcurrencyCheckAttributeConvention
DatabaseGeneratedAttributeConvention
MaxLenghtAttributeConvention
StringLenghtAttributeConvention
ColumnAttributeConvention
InversePropertyAttributeConvention
ForeignKeyPrimitivePropertyAttributeConvention
IdKeyDiscoveryConvention
AssociationInverseDiscoveryConvention
ForeignKeyNavigationPropertyAttributeConvention
OneToOneConstraintIntroductionConvention
NavigationPropertyNameForeignKeyDiscoveryConvention
PrimaryKeyNameForeignKeyDiscoveryConvention
TypeNameForeignKeyDiscoveryConvention
StoreGeneratedIdentityKeyConvention
ForeignKeyAssociationMultiplicityConvention
OneToManyCascadeDeleteConvention
ComplexTypeDiscoveryConvention
PluralizingEntitySetNameConvention
DeclaredPropertyOrderingConvention
PluralizingTableNameConvention
ColumnOrderingConvention
ColumnTypeCasingConvention
PropertyMaxLenghtConvention
DecimalPropertyConvention
ManyToManyCascadeDeleteConvention
IncludeMetadataConvention
ModelNamespaceConvention
ModelContainerConvention

Como puede observar en la lista anterior y podrá ver con más detalle en el siguiente punto dedicado a Mapping, estas convenciones son muchas y variadas. De entre ellas tenemos convenciones para especificar tipos de columnas, nombres, nombres de tablas o condiciones de relaciones entre otras. Por hablar de alguna convención en particular podríamos describir **OneToManyCascadeDeleteConvention,** convención por la cual cuando se mapee una relación de uno a muchos se establecerá una restricción de borrado en cascada. Lógicamente, a veces no nos interesará que estas convenciones existan o produzcan su efecto. Para estos casos disponemos de la posibilidad de borrarlas. Para ello, tendremos que hacer uso del método *OnModelCreating* que la

clase *DbContext* nos provee, en el siguiente ejemplo, vemos la unidad de trabajo anterior con este método sobreescrito.

```
public class CustomerContext
    : DbContext
{

// Omitido por brevedad

    protected override void OnModelCreating(DbModelBuilder
modelBuilder)
    {

    }
}
```

Como se puede observar, este método nos ofrece como parámetro un elemento de tipo *DbModelBuilder*, el cual representa el elemento central de mapeo (lo mismo que un EDM en el escenario tradicional). De entre los distintos miembros que nos ofrece esta clase destacaremos aquí la propiedad **Conventions,** propiedad que nos ofrecerá la posibilidad de revisar las distintas convenciones de las que disponemos y, opcionalmente, la posibilidad de eliminar una convención mediante el único método que ofrece: *ConventionsConfiguration*.

```
public class DbModelBuilder
{
  public virtual ConventionsConfiguration Conventions { get; }
}
```

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{

modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();
}
```

Nota: En las versiones preliminares de las betas de esta nueva versión el elemento *ConventionsConfiguration* ofrecía de forma pública métodos para incluir nuevas convenciones. Como ya se ha comentado, por falta de tiempo esta funcionalidad no está lista en esta versión.

3.5.- Mapping

Llegados a este punto ya hemos visto cómo EF 4.1 es capaz de manejar las conexiones a nuestras bases de datos así como la extensión y la configuración de sus cadenas de conexión. A mayores también hemos visto qué son las **convenciones** y la **lista de convenciones** que por defecto nos proporciona EF 4.1. Además todo esto lo

hemos visto sobre un ejemplo sencillo que nos permitió ilustrar muchos de estos conceptos. A continuación y durante todo este punto 3.5 veremos distintos escenarios de mapeo, intentando realizar los mismos casos que los que presenta el capítulo 2 de este libro y en el mismo orden. Para realizar los diferentes ejemplos de mapeos utilizaremos dos técnicas distintas: mediante atributos (en el caso en el que se pueda) y mediante el **API fluent** que EF 4.1 pone a nuestra disposición.

Nota: En **ingeniería de software**, una **interfaz fluida** o un **api fluent** (término acuñado por primera vez por **Eric Evans** y **Martin Fowler**) es una **construcción orientada a objeto** que define un comportamiento capaz de retransmitir el contexto de la instrucción de una llamada subsecuente. Generalmente, el contexto es definido a través del valor de retorno de un método llamado autorreferencial, donde el nuevo contexto es equivalente al contexto anterior terminado por medio del retorno de un contexto vacío (*void context*).

Este estilo es beneficioso debido a su capacidad de proporcionar una sensación más fluida al código, aunque algunos ingenieros encuentran el estilo difícil de leer. Una segunda crítica es que generalmente, las necesidades de programación son demasiado dinámicas para confiar en la definición estática de contacto ofrecida por una interfaz fluida.

Fuente: Wikipedia - **http://en.wikipedia.org/wiki/Fluent_interface**

3.5.1.- Las entidades

Para realizar nuestro primer ejemplo supondremos que queremos establecer el mapeo para una entidad *Customer* con la forma siguiente:

```
public class Customer
{
   public int CustomerId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
}
```

Tal y como vimos y explicamos sobre el primero de los ejemplos, la unidad de trabajo asociada no es más que cualquier clase que implemente *DbContext* y exponga un *DbSet* de esta clase, como por ejemplo podría ser la siguiente:

```
public class CRMUnitOfWork
   :DbContext
{
   public IDbSet<Customer> Customers { get; set; }

   protected override void OnModelCreating(DbModelBuilder
modelBuilder)
```

```
    {
        base.OnModelCreating(modelBuilder);
    }
}
```

Nota: En esta clase *CRMUnitOfWork* se ha sobreescrito el método *OnModelCreating* puesto que como veremos más adelante es en este dónde incluiremos características de mapeo especiales usando nuestra *Fluent API*.

Si con esta unidad de trabajo realizáramos cualquier operación, veríamos como esta entidad se mapearía con una tabla en la base de datos, concretamente con una tabla con la siguiente estructura.

```
USE [EF41_M1.CRMUnitOfWork]
GO

SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[Customers](
    [CustomerId] [int] IDENTITY(1,1) NOT NULL,
    [FirstName] [nvarchar](128) NULL,
    [LastName] [nvarchar](128) NULL,
PRIMARY KEY CLUSTERED
(
    [CustomerId] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]

GO
```

1. El descubrimiento de las claves

Lo que nos llama en primer lugar la atención es cómo se ha descubierto/asociado la clave primaria de la entidad, estableciendo además la misma como **Identity** en la tabla recién creada. La respuesta a esto es que EF 4.1 dispone de una convención llamada **IdKeyDiscoveryConvention** por la cual cualquier propiedad de nombre **Id** o bien cualquier propiedad que esté formada por el nombre de la entidad más la terminación **Id**, como es en nuestro caso, son las candidatas a ser clave primaria. Esto lo podríamos comprobar fácilmente si modificaramos el nombre de la propiedad CustomerId por XId, gracias a lo cual la infraestructura de EF nos lanzaría una excepción como la siguiente:

"System.Data.Edm.EdmEntityType: : EntityType 'Customer' has no key defined. Define the key for this EntityType."

Si bien esta convención ahorra configuración, en ocasiones es necesario que las claves de la entidad sean nombres que nada tengan que ver con el postfijo **Id**. En muchas bases de datos existentes esta nomenclatura o guía no tiene por qué estar aplicada o bien nosotros mismos no queremos aplicarla. Para ello disponemos como veremos a continuación de varias opciones.

La primera posibilidad es hacer uso de "Data Annotations" o los atributos que nos permitirán establecer elementos de mapeo directamente.

Nota: Data Annotations no es algo nuevo de EF, ni siquiera es de uso exclusivo en esta tecnología. Ya en ASP.NET Dynamic Data, por poner un ejemplo, usábamos las anotaciones para marcar ciertas características de nuestro modelo. El equipo de EF 4.1 optó por el uso de Data Annotations para marcar ciertas características de los mapeos, usando para ello los atributos existentes dónde tuviera sentido y creando nuevos atributos para aquellos elementos nuevos. Preste atención a los atributos que incluye en sus entidades puesto que si bien el namespace puede que se corresponda con System.ComponentModel.DataAnnotations en ocasiones la implementación estará en EntityFramework.dll

En el caso que nos ocupa, el atributo que nos interesa es **KeyAttribute** y su uso podría ser como el que sigue:

```
public class Customer
{
    [Key()]
    public int XId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Lógicamente, si quisiéramos aplicar varias propiedades como clave la via más natural sería incluir varios atributos de tipo *KeyAttribute*. Sin embargo, si realizamos esto obtendríamos la siguiente excepción:

"Unable to determine composite primary key ordering for type 'EF41_M1.Customer'. Use the ColumnAttribute or the HasKey method to specify an order for composite primary keys."

Seguramente, esta excepción le llamará la atención puesto que no tiene demasiado sentido el tener que especificar el orden de las claves. Sin embargo, **esto es necesario para dar soporte a algunas operaciones que podemos realizar con el nuevo API**, como por ejemplo el método de búsqueda por clave, en el que tendremos que pasar el valor de las claves en el orden en el que han sido establecidas. Para establecer este orden podemos utilizar un atributo contenido en DataAnnotations (aunque realmente

esta implementado en EntityFramework.dll) llamado **Column**, como puede verse en el siguiente fragmento de código.

```
public class Customer
{
    [Key()]
    [Column(Order=0)]
    public int XId { get; set; }
    [Key()]
    [Column(Order = 1)]
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

La propiedad de que estas columnas generadas en la base de datos sean de tipo **Identity** o no puede establecerse mediante otro atributo, en este caso llamado **DatabaseGenerated** el cual admite como parámetro la siguiente enumeración:

```
public enum DatabaseGeneratedOption
{
    //      La base de datos no genera nombres.
    None = 0,
    //
    //      La BBDD genera un valor cuando se inserta la fila.
    Identity = 1,
    //
    //      La BBDD genera un valor cuando una fila se inserta o se
actualiza.
    Computed = 2,
}
```

De esta forma, si quisiéramos que nuestra entidad *Customer* no tuviera su clave *CustomerId* de tipo identity tendríamos que aplicar este atributo como sigue:

```
public class Customer
{
    [Key()]
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

A mayores, tal y como comentábamos al principio de este punto, a mayores de la posibilidad de usar atributos para especificar detalles de mapeo, como acabamos de ver, también disponemos de un **API fluent.** En el siguiente fragmento puede verse el mapeo anterior sin necesidad de establecer atributos a nuestras clases.

```
public class CRMUnitOfWork
        :DbContext
{
```

```
    public IDbSet<Customer> Customers { get; set; }

    protected override void OnModelCreating(DbModelBuilder
modelBuilder)
    {
        modelBuilder.Entity<Customer>()
                    .HasKey(c => c.CustomerId);

        modelBuilder.Entity<Customer>()
                    .Property(c => c.CustomerId)
.HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);
    }
}
```

Si quisiéramos una clave compuesta como hicimos utilizando atributos solamente tendríamos que poner la expresión de la misma dentro de la expresión lambda utilizada en el método HasKey, tal y como podría ser la siguiente:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
                .HasKey(c =>                                          new
{ c.CustomerId,   c.XId });

}
```

Nota: Fíjese que en este caso no ha sido necesario especificar *Order*, como hicimos en el caso de atributos ya que el mismo está implícitamente establecido dentro del tipo anónimo que forma la clave.

Como habrá podido observar en estos últimos fragmentos de código, **DbModelBuilder** nos ofrece una serie de métodos para acceder tanto a las entidades como a sus elementos de una forma "fluida". Así por ejemplo *Entity<Customer>* nos permite acceder a distintos elementos (**HasKey**, **Property**, etc.) con los que configurar una entidad. En los siguientes apartados veremos más usos de estos métodos así como la aparición de alguno nuevo.

2. Sobre los nombres...

En todos nuestros ejemplos hemos dejado al sistema que seleccione tanto el nombre de las tablas como el de las columnas (por ahora nunca hemos partido de una base de datos existente y no se nos habría presentado esta casuística). Sin embargo en no pocas ocasiones tendremos que poder seleccionar como se llaman/llamarán las tablas y las columnas que las forman. Esto será así bien porque el mapeo lo estemos haciendo contra una base de datos existente, o bien porque el modelo relacional tiene una guía de nomenclatura que tenemos que seguir. Como verá a continuación, el establecimiento de

los nombres de tablas y columnas es realmente sencillo haciendo uso del conocido atributo **ColumnAttribute** y un nuevo atributo llamado **TableAttribute.**

```
[Table("tbCustomers")]
public class Customer
{
    [Column("cId")]
    public int CustomerId { get; set; }
    [Column("cFName")]
    public string FirstName { get; set; }
    [Column("cLName")]
    public string LastName { get; set; }
}
```

Por supuesto, de igual forma que lo podemos hacer por medio de atributos, lo mismo podríamos hacer usando el API fluent. Fíjese en el uso de dos nuevos métodos, **ToTable** y **HasColumnName**.

```
public class CRMUnitOfWork
        :DbContext
{
   public IDbSet<Customer> Customers { get; set; }

   protected override void OnModelCreating(DbModelBuilder
modelBuilder)
   {
      modelBuilder.Entity<Customer>()
                  .ToTable("tbCustomers");

      modelBuilder.Entity<Customer>()
                  .Property(c => c.CustomerId)
                  .HasColumnName("cId");

      modelBuilder.Entity<Customer>()
                  .Property(c => c.FirstName)
                  .HasColumnName("cFName");

      modelBuilder.Entity<Customer>()
                  .Property(c => c.FirstName)
                   .HasColumnName("cLName");
   }
}
```

3. Los tipos, opcionales y obligatorios

Otro de los elementos comunes que tendremos que establecer para nuestros mapeos de entidades con tablas de la base de datos es si los elementos son opcionales o no, es decir, si las columnas permiten nulos para ciertos valores o no. Por convención EF establece como *nulables* a todas las columnas de tipo **string** y **byte[]**, y como no *nulables* al resto de tipos primitivos, siempre y cuando éstos no se hayan convertido a *Nullable<T>*. En la siguiente tabla podemos ver cuál sería la asignación por defecto de tipos y *nulabilidad* para los principales tipos de datos en .NET

Tipo C#	Tipo	¿Admite nulo por defecto?
String	Nvarchar(max)	Si
Int32	Int	No
Int64	BigInt	No
Single	Real	No
Double	Float	No
Decimal	Decimal(18,2)	No
Datetime	DateTime	No
Byte[]	Varbinary(max)	Si
TimeSpan	Time(7)	No

Nota: Fíjese como las asignaciones de tipos son realmente las asignaciones de compatibilidad entre los tipos de datos de Sql Server y .NET. Puede obtener más información sobre esta asignación de compatibilidad en el siguiente enlace:

http://msdn.microsoft.com/es-es/library/ms131092(v=sql.90).aspx

El caso del tipo de datos decimal es quizás el más especial de todos porque conlleva una convención asociada, llamada *DecimalPropertyConvention* en virtud de la cual, el valor de precisión y escala (18,2) pueden ser modificados si se especifican los *facets* necesarios, aunque esto solamente es posible utilizando el API fluent y no mediante atributos. Lo veremos más detenidamente en el siguiente apartado.

Si en algún momento nos interesara cambiar la asignación de tipo por convención podríamos utilizar una de las propiedades del ya conocido atributo **Column** llamada **TypeName**. Con ella podemos establecer el nombre del tipo al que queremos mapear la columna. A continuación vemos como establecer una columna que por defecto sería **int** a **bigint**.

```
[Column(TypeName="bigint")]
public int Age { get; set; }
```

Nota: Uno de los tipos de dato que suele provocar más quebraderos de cabeza es el tipo *DateTime*. En Sql Server este tipo de datos .NET puede mapearse a *datetime* y *datetime2* . La principal diferencia entre ellos es el rango de soporte. Mientras que *datetime2* soporta desde el 0001/01/01 hasta la eternidad, el *epoch* de *datetime* comienza en 1753. Por esta razón, si intenta guardar una entidad con un campo *datetime* y el valor de éste por defecto (01/01/0001) podría obtener una excepción como la siguiente:

"The conversion of a datetime2 data type to a datetime data type resulted in an

out-of-range value. The statement has been terminated"

Para resolverlo asegúrese de que el valor de esa propiedad está dentro del rango de *DateTime* o bien establezca en el mapeo de la misma que la propiedad subyacente es datetime2.

El cambio de asignación de nulos o no en una columna también se puede modificar de acuerdo a su comportamiento por defecto. Así, si quisiéramos que una columna de un tipo primitivo admitiera nulos solamente tendríamos que establecer este tipo como un tipo **Nullable<T>.**

```
[Column(TypeName="bigint")]
public int? Age { get; set; }
```

Para el caso contrario, es decir, cuando tenemos un tipo como *string* que establece la columna como *nulable* por defecto, también podemos modificar el comportamiento. Para ello el atributo **RequiredAttribute** nos permite establecer como no nulable una columna.

```
[Required()]
public string LastName { get; set; }
```

Por supuesto, este trabajo también se puede hacer por medio de un **API fluent** tal y como podemos observar en el siguiente fragmento, observe los nuevos métodos **IsRequired** y **IsOptional**.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   modelBuilder.Entity<Customer>()
               .HasKey(c => c.CustomerId);

    modelBuilder.Entity<Customer>()
                .Property(c => c.Age)
                .HasColumnType("bigint")
                .IsRequired();

    modelBuilder.Entity<Customer>()
                .Property(c => c.LastName)
                .IsOptional();

}
```

4. Los tipos, la precisión y otros facets

En el apartado anterior hemos visto cuál es la asignación de tipos por defecto, y para aquellos que admiten precisión, como *nvarchar* y *decimal*, cuál es la asignación que se le otorga. De hecho, en una nota comentamos como *decimal* admite un cambio

de precisión y escala, aunque no mediante un atributo. En las siguientes líneas, veremos cómo hacer los cambios en la precisión y escala de los tipos así como la configuración de otras propiedades.

Para el tipo de datos **String** disponemos de otro atributo nuevo contenido en **Data Annotations**, llamado **MaxLength**. Este nuevo atributo, como vemos a continuación, nos permitirá de una forma sencilla establecer el tamaño de nuestra cadena de caracteres.

```
[MaxLength(256)]
public string FirstName { get; set; }
```

Nota: Tenga cuidado con el uso de MaxLength en tipos que no admiten este elemento como *int*, *double*, etc. Si bien el mapeo no lanzará ninguna excepción puede que tenga comportamientos inexperados en los procesos de validación de las entidades.

Opcionalmente también podemos utilizar el atrituto **StringLength** para especificar el tamaño de la cadena de caracteres. Se deja al lector seleccionar el atributo que desee para esta tarea.

```
[StringLength(400)]
public string FirstName { get; set; }
```

Estoy seguro que al ver estos dos atributos se le ha venido a la cabeza el uso de **MinLengthAttrubute** o la propiedad **MinimunLength** de **StringLength**. Pues bien, éstos también pueden ser usados, aunque, solamente tendrán efecto de cara a la validación de los valores de las entidades, nada con respecto al mapeo y por lo tanto, esto será tratado más adelante.

Como siempre, el mismo trabajo que hacemos con los atributos podemos hacerlo con nuestra **API fluent**. A continuación vemos nuestro método *OnModelCreating* con algunos ejemplos de uso del API fluent para especificar la precisión y escala de algunas propiedades:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   modelBuilder.Entity<Customer>()
               .Property(c => c.FirstName)
               .HasMaxLength(400);

   modelBuilder.Entity<Customer>()
               .Property(c => c.LastName)
               .HasMaxLength(200);
}
```

A mayores, con nuestra API también podemos establecer que estos elementos sean de longitud fija (se traducirá en un campo **nchar** en vez de **nvarchar**) por medio del método **IsFixedLength**.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
                .Property(c => c.FirstName)
                .HasMaxLength(400)
                .IsFixedLength();

    modelBuilder.Entity<Customer>()
                .Property(c => c.LastName)
                .HasMaxLength(200)
                .IsFixedLength();
}
```

Por medio de nuestro API también podemos establecer algunos *facets* que no tenemos disponibles por medio del mapeo con atributos, como es el caso del tipo **unicode** en nuestras cadenas de caracteres, observe el uso de **IsUnicode**:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
                .Property(c => c.FirstName)
                .HasMaxLength(400)
                .IsUnicode();

}
```

Nota: Realmente, en la CTP 5 si podríamos llegar a construir convenciones para establecer por medio de atributos *facets* como el de Unicode. Sin embargo, puesto que en la reléase del producto han eliminado la posibilidad de realizar convenciones personalizadas esta posibilidad ya no está disponible.

Si recuerda, unos párrafos atrás comentábamos que el tipo **decimal** admitía una convención, **DecimalPropertyConvention**, por la cual la precisión y la escala se establecían a 18 y 2 respectivamente si no se establecían estos elementos en el mapeo. Además comentamos también que estos dos elementos no los podemos establecer por medio de atributos: hay que recurrir al API directamente. A continuación, en el siguiente fragmento de código puede ver el uso del método **HasPrecision**, gracias al cual podremos establecer las facets precisión y escala:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
                .Property(c => c.Rate)
                .HasPrecision(20, 10);

}
```

5. La concurrencia, ¿Dónde está nuestro ConcurrencyModel.Fixed?

Seguramente, si usted está haciendo un repaso mental con respecto a las diferencias de mapeo que teníamos usando **Entity Data Model** y ahora, quizás esté echando en falta la propiedad **ConcurrencyMode** que podíamos establecer en nuestros **EDM**. Pues bien, por suerte, la posibilidad de establecer que propiedades formarán parte de la comprobación de concurrencia también están disponibles, tanto mediante el uso de atributos como lógicamente mediante nuestro api fluent.

El atributo **ConcurrencyCheck** nos permite establecer la propiedad o las propiedades que formarán parte de la verificación de concurrencia optimista. En el siguiente fragmento puede ver un ejemplo en el que las propiedades *FirstName* y *LastName* de la clase *Customer* serán tenidas en cuenta para gestionar la concurrencia:

```
public class Customer
{
   public int CustomerId { get; set; }

   [ConcurrencyCheck()]
   public string FirstName { get; set; }

   [ConcurrencyCheck()]
   public string LastName { get; set; }

}
```

Si quisiéramos hacer esto directamente con nuestro API solamente tendríamos que escribir lo siguiente (fíjese en el método **IsConcurrencyToken**):

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   modelBuilder.Entity<Customer>()
               .Property(c => c.FirstName)
               .IsConcurrencyToken(true);

   modelBuilder.Entity<Customer>()
               .Property(c => c.LastName)
               .IsConcurrencyToken(true);
}
```

6. Versionado de filas rowversion o version-stamping

Además de los chequeos de concurrencia, aplicables a cualquiera de las propiedades de nuestras entidades, EF 4.1 nos proporciona la posibilidad de utilizar un mecanismo de *RowVersion* en base de datos, si éste está disponible en el proveedor.

Básicamente, este mecanismo consiste en utilizar un tipo de datos, en Sql Server **timestamp** (puede que lo conozca más familiarmente como **rowversion**, un sinónimo de este tipo), que generalmente nos proporciona elementos de generación automática que se actualizan cada vez que una fila se inserta o actualiza en una tabla.

Nota: Puede obtener más información sobre el tipo, timestamp, y su uso en la siguiente dirección:

http://msdn.microsoft.com/en-us/library/ms182776(SQL.90).aspx

Con este tipo de datos, mapeado tal y como veremos a continuación, podremos garantizar que las operaciones se realizan para una versión específica de una fila, y por lo tanto obtener y manejar posibles problemas de concurrencia. Con el fin de ver un ejemplo de uso, partiremos de la siguiente versión modificada de nuestra entidad *Customer*:

```
public class Customer
{
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    [Timestamp()]
    public byte[] RowVersion { get; set; }

}
```

En primer lugar, sobre la definición de la entidad anterior, note como la propiedad de nombre *RowVersion* se ha establecido como una propiedad de tipo **byte[],** de forma obligatoria para Sql Server, ya que el tipo **timestamp** es en esencia un conjunto de 8 bytes. Entity Framework 4.1, automáticamente, al ver el atributo de la propiedad, establece en la base de datos el tipo timestamp para esta columna:

```
CREATE TABLE [dbo].[Customers](
    [CustomerId] [int] IDENTITY(1,1) NOT NULL,
    [FirstName] [nvarchar](max) NULL,
    [LastName] [nvarchar](max) NULL,
    [RowVersion] [timestamp] NOT NULL,
PRIMARY KEY CLUSTERED
(
    [CustomerId] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]
```

A mayores, la infraestructura de trabajo de EF hace uso de esta propiedad para que sea tomada en cuenta en las operaciones de actualización y borrado, de tal forma, que al igual que se hace con las propiedades de tipo **ConcurrencyCheck**, su valor sea comprobado para verificar que la fila en cuestión no se ha modificado/borrado previamente en la base de datos. Por lo tanto, por ejemplo, en un proceso de actualización de esta entidad podríamos encontrarnos una consulta *update* como la siguiente:

```
exec sp_executesql N'update [dbo].[Customers]
set [FirstName] = @0
where (([CustomerId] = @1) and ([RowVersion] = @2))
select [RowVersion]
from [dbo].[Customers]
where @@ROWCOUNT > 0 and [CustomerId] = @1',N'@0 nvarchar(max) ,@1
int,@2 binary(8)',@0=N'unai',@1=1,@2=0x00000000000007D2
```

Note como al igual que con los campos de tipo *identity*, después de la operación se recupera el valor asignado a la propiedad **RowVersion** con el fin de que se siga teniendo en cuenta para futuras operaciones. Lógicamente, el mapeo mediante nuestra API fluent también es realmente sencillo. A continuación, en el siguiente fragmento de código, puede ver este mismo caso. Fíjese en la utilización del método **IsRowVersion**:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   modelBuilder.Entity<Customer>()
               .Property(p => p.RowVersion)
               .IsRowVersion();
}
```

7. Bien, pero, ¿y si esto no quiero que forme parte de mi mapeo?

En ocasiones, cuando creamos nuestro modelo de entidades (es decir, nuestro dominio) necesitamos que estas entidades puedan contener propiedades que no necesariamente tengan que estar asociadas con columnas u otros elementos en la base de datos. Hasta ahora, utilizando EDM con cualquier plantilla de generación de código, la solución que teníamos era el uso de clases parciales. Ahora tenemos una solución más elegante que la de tener que crear un nuevo fichero de código para realizar esta tarea.

El atributo **NotMappedAttribute**, también definido en *System.ComponentModel.DataAnnotations* nos permite especificar qué propiedades o tipos no deseamos que se mapeen. En el siguiente fragmento podemos ver una propiedad *FullName* decorada con este atributo.

```
public class Customer
{
   //Omited for brevity

   [ConcurrencyCheck()]
   public string FirstName { get; set; }

   [ConcurrencyCheck()]
   public string LastName { get; set; }

   [NotMapped()]
   public string FullName
   {
      get
      {
         return string.Format("{0}, {1}", this.LastName,
this.FirstName);
```

```
        }
    }
}
```

En el caso en el que prefiera utilizar directamente el API, el método **Ignore** realiza esta tarea de forma idéntica:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
                .Ignore(c => c.FullName);
}
```

8. Propiedades de solo lectura

Aunque parezca un concepto a priori sencillo, en realidad, las propiedades de solamente lectura tienen más "intríngulis" de lo que parece (si no ha trabajado ya con NHibernate, que dispone del mismo concepto). Cuando uno se refiere a una propiedad de sólo lectura, se refiere a una propiedad cuyo valor es calculado en el dominio pero no se puede establecer, "*setear*" por decir un anglicismo comprensible. Sin embargo, el valor de esta propiedad sí se almacena en la base de datos.

De entrada, uno pensaría en algo similar a lo siguiente:

```
public class Customer
{
   public int CustomerId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }

   public string FullName
   {
      get
      {
         return string.Format("{0},{1}", this.LastName,
this.FirstName);
      }
   }
}
```

Sin embargo, aunque técnicamente es una propiedad de solamente lectura, esto no es lo que buscamos, puesto que la misma no se mapearía y por lo tanto su valor no podría guardarse en la base de datos, no existiría la columna.

Para realizar nuestro trabajo solamente tendríamos que agregar un método **set** vacío, tal y como sigue:

Nota: Si, entiendo que incluir un método set vacío no es muy elegante. En NH se puede establecer un atributo para esto, "access=readonly".

```
public class Customer
{
   public int CustomerId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }

   public string FullName
   {
      get
      {
         return string.Format("{0},{1}", this.LastName,
this.FirstName);
      }
      set { }

   }
}
```

Con esto, si agregáramos una nueva entidad con los valores de nombre "Unai" y apellidos "Zorrilla Castro", en la columna *fullname* de la base de datos veríamos algo como lo siguiente:

Results | Messages

	CustomerId	FirstName	LastName	FullName
1	1	Unai	Zorrilla Castro	Zorrilla Castro,Unai

Figura 4.- Valor de la columna de una propiedad de solo lectura

9. Todo en OnModelCreating? Buff, no disponemos de otra posibilidad?

Creo que no perdería una apuesta si dijese que muchos de los lectores habrán pensado que establecer los mapeos con el API fluent dentro de nuestro *OnModelCreating* podría volverse algo inmanejable en cuanto tuviésemos un buen número de entidades a configurar. Efectivamente, este caso podría darse. Sin embargo el grupo de EF nos provee de una alternativa para solucionar y aligerar este problema.

Cuando nosotros en el código de mapeo escribimos la llamada a **Entity<Customer>**, por poner un ejemplo, este método nos devuelve una clase de tipo **EntityTypeConfiguration<TEntity>,** clase que después utilizamos para configurar nuestra entidad. Si echásemos un pequeño vistazo a las propiedades que nos ofrece **DbModelBuilder** veríamos como esta dispone de una propiedad llamada **Configurations** dónde podemos agregar diferentes elementos de tipo **EntityTypeConfiguration**. Es decir, podemos agregar las distintas configuraciones de entidades que tengamos. Esta infraestructura está creada para que nosotros podamos separar la configuración del método *OnModelCreating*, estableciendo la misma por medio de diferentes instancias de **EntityTypeConfiguration<TEntity>**.

Veamos un ejemplo en el que intentaremos separar la configuración de nuestra clase *Customer*. Para ello crearemos una nueva clase, que llamaremos *CustomerEntityTypeConfiguration*, como muestra el siguiente fragmento:

```
class CustomerEntityTypeConfiguration
        :EntityTypeConfiguration<Customer>
{
    public CustomerEntityTypeConfiguration()
    {
        this.HasKey(p => p.CustomerId);

         this.Property(p => p.FirstName)
             .HasMaxLength(400);
    }
}
```

Al ser de tipo *EntityConfiguration* podemos realizar sobre su instancia, **this**, las mismas operaciones que realizábamos anteriormente. Lógicamente, una vez creada la configuración solamente tendremos que agregarla a la lista de configuraciones.

```
public class CRMUnitOfWork
        :DbContext
{
    public IDbSet<Customer> Customers { get; set; }

    protected override void OnModelCreating(DbModelBuilder
modelBuilder)
    {
        modelBuilder.Configurations.Add(new
CustomerEntityTypeConfiguration());

    }
}
```

Con esto, ya habrá visto como separar de una forma sencilla y potente todo el código de mapeo de las entidades que forman parte del dominio. De forma general, el uso de estos elementos de configuración será la forma habitual de trabajo, aunque para el caso de demos y de pruebas rápidas se recurra a realizar el mapeo directamente desde el método *OnModelCreating*.

3.5.2.- Tipos complejos

En el capítulo 2, punto 1.1.1, además de las entidades también comentamos la existencia de tipos complejos, y como, esta característica introducida en Entity Framework 4.1 nos permite realizar o implementar modelos de dominio más acordes a nuestro lenguaje. Por suerte, ahora en Code First también tenemos disponible esta posibilidad de mapeo y como hemos visto hasta ahora, su uso es realmente sencillo. Para ver el uso y el mapo de tipos complejos partiremos de una entidad *Customer* que contiene ciertas propiedades, City, *ZipCode* y *Street*, que refactorizaremos en un nuevo tipo, *Address*.

```
public class Customer
{
   public int CustomerId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string City { get; set; }
   public string ZipCode { get; set; }
   public string Street { get; set; }
}
```

```
public class Customer
{
   public int CustomerId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public Address Address { get; set; }
}
public class Address
{
   public string City { get; set; }
   public string ZipCode { get; set; }
   public string Street { get; set; }
}
```

Sin hacer ningún trabajo adicional, solamente usando una unidad de trabajo tal cual se ha venido definiendo en los puntos anteriores, las convenciones de EF se encargarán de realizar nuestro mapeo con la base de datos. La convención por defecto en este caso establece que el nombre de las columnas de la tabla de mapeo con la entidad *Customer* que representan al tipo complejo, siguen la estructura **[NombreTipoComplejo]_[NombrePropiedad]**. Por lo tanto el esquema de la tabla creada sería algo similar a:

```
CREATE TABLE [dbo].[Customers](
    [CustomerId] [int] IDENTITY(1,1) NOT NULL,
    [FirstName] [nvarchar](128) NULL,
    [LastName] [nvarchar](128) NULL,
    [Address_City] [nvarchar](128) NULL,
    [Address_ZipCode] [nvarchar](128) NULL,
    [Address_Street] [nvarchar](128) NULL,
PRIMARY KEY CLUSTERED
```

```
(
    [CustomerId] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON)
ON [PRIMARY]
) ON [PRIMARY]

GO
```

Por supuesto, al igual que hacíamos con las entidades, podemos modificar ciertos parámetros de mapeo directamente en nuestro método *OnModelCreating*, o bien por medio de un elemento de configuración separado. A continuación en el siguiente fragmento de código se puede ver el uso de un nuevo método, **ComplexType**, contenido en la clase **DbModelBuilder**, que al igual que **Entity** para entidades nos permitirá realizar la configuración de mapeo de un tipo complejo.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   modelBuilder.ComplexType<Address>()
               .Property(p => p.City)
               .HasColumnName("City");

   modelBuilder.ComplexType<Address>()
               .Property(p => p.Street)
               .HasColumnName("Street");

  modelBuilder.ComplexType<Address>()
              .Property(p => p.ZipCode)
              .HasColumnName("ZipCode");
}
```

Nota: ComplexType ofrece las mismas posibilidades de configuración que *Entity<>()*.

En el caso de querer separar nuestra configuración del método *OnModelCreating* tendremos que recurrir a la creación de una nueva clase heredando de **ComplexTypeConfiguration<TComplexType>**. Recuerde que para entidades el tipo del que debíamos hacer una subclase era **EntityTypeConfiguration<TEntity>.**

```
class AddressComplexTypeConfiguration
       :ComplexTypeConfiguration<Address>
{
   public AddressComplexTypeConfiguration()
   {
       this.Property(p => p.ZipCode)
           .HasColumnName("ZipCode");

       this.Property(p => p.Street)
           .HasColumnName("Street");

       this.Property(p => p.City)
```

```
            .HasColumnName("City");
    }
}
```

Nota: Por supuesto, tal y como explicamos anteriormente, una vez que hemos creado la clase de configuración ésta tiene que ser incluída dentro de la colección de configuraciones que nos ofrece el parámetro *modelBuilder* dentro del método *OnModelCreating*.

El mapeo por medio de *ComplexProperty* no es obligatorio, en realidad se podrían configurar las propiedades de un tipo complejo por medio de una entidad.

A continuación puede ver un ejemplo de ello:

```
modelBuilder.Entity<Customer>
                    .Property(c=>c.Address.City)
                    .HasColumnName("City")
```

Tal y como acabamos de ver, EF 4.1 pone a nuestra disposición la posibilidad de marcar de una forma declarativa que estamos trabajando con un tipo complejo, por medio del método **ComplexType** o bien con la clase **ComplexTypeConfiguration**. Sin embargo tal y como vimos en el primer caso de tipo complejo esto no es necesario. En ocasiones, si no hacemos uso de una configuración específica de tipo complejo, es posible que la plataforma no pudiera diferenciar éste de una asociación 1..0,1. Imagine el siguiente escenario:

```
public class Customer
{
   public int CustomerId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public Address Address { get; set; }
}
public class Address
{
   public int Id { get; set; }
   public string City { get; set; }
   public string ZipCode { get; set; }
   public string Street { get; set; }
}
```

Con esta definición de las entidades *Customer* y *Address*, EF no sabría distinguir nuestra intencionalidad con respecto a *Address*, es decir, si *Address* es un tipo complejo o bien una asociación. Realmente, tal y como está, con *Address* con una propiedad de

nombre *Id* (recuerde como por convención se averiguan las claves) entenderá esto como una asociación. Si no queremos esto y no "hacemos de forma declarativa el mapeo de tipo complejo", podemos recurrir a una anotación llamada **ComplexType**, por la cual indicaremos que *Address* es un tipo complejo.

```
public class Customer
{
   public int CustomerId { get; set; }

   public string FirstName { get; set; }

   public string LastName { get; set; }

   public Address Address { get; set; }
}

[ComplexType()]
public class Address
{
   public int Id { get; set; }

   public string City { get; set; }

   public string ZipCode { get; set; }

   public string Street { get; set; }
}
```

3.5.3.- Asociaciones

Como no cabía esperar otra cosa, EF 4.1 y Code First soportan todo el rango de asociaciones y cardinalidades que están soportados en EDM directamente. Desde relaciones con cardinalidades 0,1 o * hasta asociaciones uni-direccionales y bi-direccionales, pasando -como no- por los distintos tipos de asociaciones que tenemos en EF: asociaciones independientes y asociaciones de clave externa.

Nota: Si algo de lo comentado en este párrafo no le suena o quiere repasar los conceptos en el capítulo 2, página 25 del libro, se detallan con exactitud todos estos elementos.

Cuando se habla de asociaciones uni-direccionales y bi-direccionales no se está especificando nada más que si el grafo es completo o no. Es decir, si tenemos una relación de clientes a pedidos, si en cada pedido existe una relación al cliente entonces es bi-direccional, en caso contrario se dice que es uni-direccional.

1. Asociaciones one-to-one

Al igual que pasó a lo largo del capítulo 2 nos gustaría decir que algunos de los ejemplos que iremos viendo ahora y a lo largo del resto del texto, puede que no tengan demasiado sentido en situaciones de negocio reales. Por lo tanto, no tenga tanto en cuenta los casos como los fines ilustrativos y didácticos de los mismos.

Para este primer ejemplo supondremos que queremos realizar una asociaciones uno a uno entre las entidades *Customer* y *Address*. Además esta asociación la deseamos hacer bidireccional. La estructura de las entidades es tal cual se ve a continuación:

```
public class Customer
{
   public int CustomerId { get; set; }

   public string FirstName { get; set; }

   public string LastName { get; set; }

   public Address Address { get; set; }
}

public class Address
{
   public int AddressId { get; set; }

   public string City { get; set; }

   public string ZipCode { get; set; }

   public string Street { get; set; }

   public Customer Customer { get; set; }

}
```

Si solamente dejamos esta definición de entidades y estableciéramos una unidad de trabajo como sigue:

```
public class CRMUnitOfWork
       :DbContext
{
   public IDbSet<Customer> Customers { get; set; }

    public IDbSet<Address> Address { get; set; }

    protected override void OnModelCreating(DbModelBuilder
modelBuilder)
    {

    }
}
```

Si intentáramos utilizar esta unidad de trabajo directamente veríamos una excepción similar a:

"Unable to determine the principal end of an association between the types...."

La razón de esta excepción es que, por defecto, no es posible determinar cómo es la relación entre las entidades *Customer* y *Address*. Piense que en realidad no existen relaciones 1..1 (por lo menos no sin introducir un ciclo de restricciones, algo que algunos motores como Sql Server no admiten), y por lo tanto dentro de una relación 1..0,1 alguno debe ejercer la tarea de principal, algo que no se ha establecido en nuestro caso. Para solventar este problema, como se imaginará, tendremos que recurrir a indicar esto mediante mapeo, por anotaciones o de forma imperativa con nuestro API fluent. Como hasta ahora siempre hemos empezado mediante anotaciones en este caso no seremos menos. El atributo **ForeignKeyAssociationAttribute** nos permitirá especificar que propiedades son las claves externas de una relación, especificando mediante una cadena de caracteres la propiedad de navegación asociada a esta relación.

Usando este atributo, podríamos realizar el mapeo de forma declarativa como sigue:

```
public class Customer
{
    public int CustomerId { get; set; }

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public virtual Address Address { get; set; }
}

public class Address
{
   [ForeignKey("Customer")]
   public int AddressId { get; set; }

   public string City { get; set; }

   public string ZipCode { get; set; }

   public string Street { get; set; }

   public virtual Customer Customer { get; set; }

}
```

Nota: Puesto que seguro que lo está pensando: sí, es una lástima que desde el equipo de ADO.NET no hubieran propuesto una sobrecarga del atributo en el que pudiéramos indicar una expresión lambda en vez de un conjunto de caracteres para indicar la propiedad de navegación. Generalmente este tipo de cosas suele provocar bastantes errores difíciles de depurar por su carácter no tipado. En su descargo, *ForeignKeyAttribute* puede tener un comportamiento distinto en función de donde se aplique, puesto que se puede realizar en la propiedad de clave externa o bien en la propiedad de navegación inversa.

El uso de **virtual** dentro de las asociaciones se fundamenta por la misma razón que expusimos en este mismo libro con respecto a los objetos POCO, es decir, para facilitar la generación de proxies dinámicos. Revise el capítulo dedicado a este tema para obtener más información.

Ahora, si utilizásemos nuestra unidad de trabajo, el esquema de base de datos con el que trabajaríamos sería similar al siguiente:

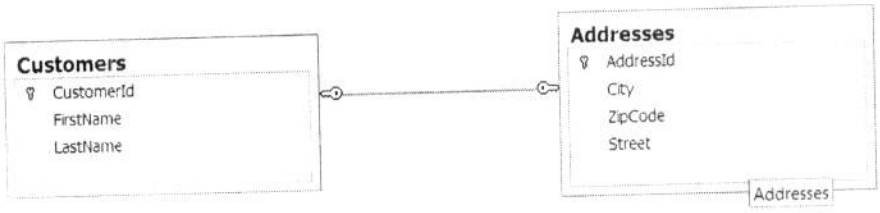


Figura 5.- Diagrama One-To-One

Si quisiéramos realizar este mismo trabajo sin el uso de atributos tendríamos que utilizar alguno de los métodos que nos permiten, a partir de una configuración de entidad, establecer una asociación. En la siguiente tabla puede ver la lista de métodos recién comentada:

Tabla 8.- Métodos de configuración de asociaciones

Método	Descripción
HasMany	Permite configurar una asociación a muchos (1..*). En este caso no aplica
HasOptional	Permite configurar una asociación a un elemento opcional (1..0,1).
HasRequired	Permite configurar una asociación a un elemento. (1..1)

Para nuestro caso, el trabajo a realizar sería el siguiente:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   modelBuilder.Entity<Customer>()
               .HasOptional(c => c.Address)
               .WithRequired(c => c.Customer);
}
```

Le recomiendo que vea como cada uno de los métodos de la tabla anterior nos ofrece distintas opciones en función del tipo de asociación que queramos realizar.

Además jugar con las distintas combinaciones que se ofrecen le puede ayudar a entender todas las posibilidades de mapeo que tenemos (required:required, required:optional, optional:optional).

Nota: En el caso de no querer tener que especificar las navegaciones inversas, tanto *WithRequired* como *WithOptional* nos ofrecen una sobrecarga en la que no es necesario pasar la expresión de la navegación inversa, puesto que no existirá en las entidades, pero si la relación en la base de datos.

2. Asociaciones one-to-many

Como no podía ser de otra forma EF 4.1 nos permite trabajar con relaciones uno a muchos de una forma muy natural, tanto por medio de convenciones capaces de descubrirnos las relaciones y sus inversas como por medio de mapeos explícitos.

El primer ejemplo que veremos será un caso sencillo de *Customer-Order* con una relación de uno a muchos.

```
public class Customer
{
    public int CustomerId { get; set; }

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public virtual ICollection<Order> Orders { get; set; }
}

public class Order
{
    public int OrderId { get; set; }

    public decimal TotalOrder { get; set; }

    public DateTime OrderDate { get; set; }
}
```

En este código, deberemos anotar dos elementos. El primero se refiere al tipo de las propiedades de navegación, que al igual que pasa con POCO éstas tienen que ser de tipo *ICollection* (otra vez para favorecer la generación de *proxies* de entidades, revise el punto *2.7, "Entidades POCO y la gestión del estado"* para obtener más información). El segundo de los elementos es que las relaciones no tienen por qué ser bidireccionales y ni siquiera tienen que ser asociaciones de clave externa, como por ejemplo es este caso. En unos pocos párrafos veremos cómo dependiendo de esto el sistema introduce en nuestros mapeos algunas restricciones o no. Ahora, si utilizáramos una unidad de trabajo con un **DbSet** para *Customer*, el mapeo que se realizaría sería tal cual el que podemos ver en la siguiente figura, figura 6.

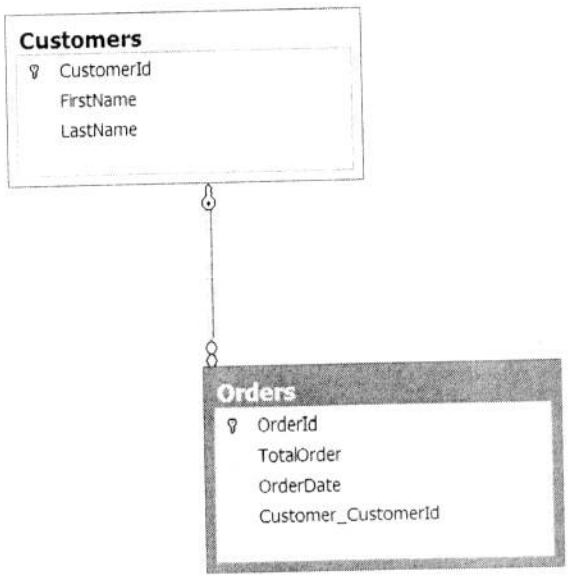


Figura 6.- Mapeo de una relación uno a muchos

Nota: La unidad de trabajo no tendría por qué tener un DbSet para nuestra entidad *Order*. De hecho, en soluciones orientadas al dominio generalmente solamente se crearían estos elementos para los agregados del modelo

Lógicamente, si nos interesa tener una relación bidireccional (con o sin asociación de clave externa) también podríamos hacerlo, en el siguiente fragmento podemos ver la entidad **Order** modificada para incluir la relación inversa.

```
public class Order
{
   public int OrderId { get; set; }

   public decimal TotalOrder { get; set; }

   public DateTime OrderDate { get; set; }

   public int CustomerId { get; set; }

   public virtual Customer Customer { get; set; }
}
```

Nota: En el caso de haber seleccionado una asociación independiente, es decir, sin la propiedad *CustomerId* en nuestra entidad *Order*, éste se habría inferido en la base de datos con un nombre por convención ([NombreEntidad]_[NombreClavePrimariaInversa]). En nuestro caso hubiera generado una columna con nombre *Customer_CustomerId*. Si estuviéramos adaptándonos a un modelo relacional existente podríamos configurar esto como sigue:

```
modelBuilder.Entity<Customer>()
            .HasMany(c => c.Orders)
            .WithRequired(o => o.Customer)
            .Map(mc => mc.MapKey("cid"));
```

Si vuelve a revisar la estructura del modelo relacional creado, figura 7, podrá ver como ahora, lógicamente además de utilizar la propiedad *CustomerId* como la clave externa de la relación, la misma tiene asignada una regla de "borrado en cascada", figura 8.

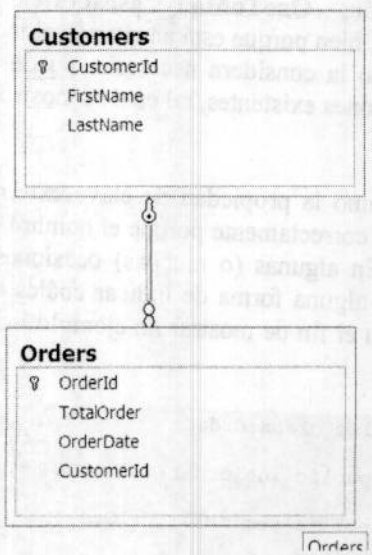


Figura 7.- Esquema de asociación con clave externa

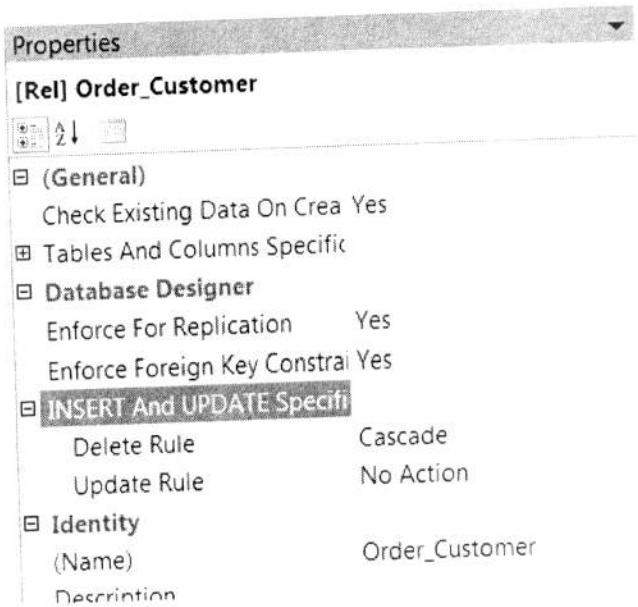


Figura 8.- Regla de borrado en cascada para la relacion

Nota: El establecimiento de la regla de "borrado en cascada" se establece por convención, **OneToManyCascadeDeleteConvention**. Si usted no quiere utilizarla, bien porque está adaptando el mapeo a una base de datos existente o bien porque no la considera necesaria, puede eliminar esta convención de la lista de convenciones existentes, tal cual se mostro anteriormente

Fíjese como la propiedad de clave externa, *CustomerId*, definida en *Order* se ha interpretado correctamente porque el nombre de la misma se correspone con la clave de *Customer*. En algunas (o muchas) ocasiones, esto no será posible y tendremos que disponer de alguna forma de indicar cuáles de las propiedades representan a la clave externa. Con el fin de mostrar un ejemplo imágine que ahora la entidad *Order* es tal y como sigue:

```
public class Order
{
   public int OrderId { get; set; }

   public decimal TotalOrder { get; set; }

   public DateTime OrderDate { get; set; }

   public int CID { get; set; }

   public virtual Customer Customer { get; set; }
}
```

En este caso, puesto que EF no es capaz de "ver" qué propiedad tiene la intencionalidad de ser clave externa, creará (con un nombre por convención) una columna en la tabla *Order* para establecer esta relación. Lógicamente, la propiedad *CID* no será materializada con estos valores. Para forzar el uso de una propiedad como clave externa, usando anotaciones, podemos recurrir al atributo **ForeignKeyAttribute**, el cual nos permite establecer como parámetro el nombre de la propiedad que funcionará de clave externa.

A continuación podemos ver el uso de este atributo.

```
public class Order
{
   public int OrderId { get; set; }

   public decimal TotalOrder { get; set; }

   public DateTime OrderDate { get; set; }

   public int CCId { get; set; }

   [ForeignKey("CCId")]
   public virtual Customer Customer { get; set; }
}
```

Por desgracia, hay escenarios dónde el uso del atributo *ForeignKey* no nos servirá, y, en algunos casos no nos quedará más remedio que acudir a realizar el mapeo mediante nuestra API fluent.

Nota: Creo que somos muchos los que en cierta manera nos alegramos de esto, es decir, de tener que recurrir a mapeos mediante el API sin anotaciones. El porqué es porque realmente con esto libramos a nuestras entidades de todo lo que tenga que ver con EntityFramework.dll y por lo tanto promocionamos la ignorancia de la persistencia. Recuerde aquí, tal y como hemos dicho ya varias veces, que a pesar de que todas las anotaciones están definidas en el namespace System. ComponentModel.DataAnnotations, muchas están realmente implementadas, y por lo tanto estamos obligados a referenciar, en Entity Framework.dll.

Para presentar un ejemplo de esto último supongamos que queremos implementar un escenario dónde tenemos un usuario y mensajes, de tal forma que un usuario puede ver una lista de sus mensajes enviados y de los mensajes para los cuales es el receptor.

El código de las entidades del escenario anterior podría ser algo similar a lo siguiente:

```
public class User
{
   public virtual int UserId { get; set; }
   public virtual string FirstName { get; set; }
   public virtual string LastName { get; set; }
   public virtual string Email { get; set; }
```

```
    public virtual string Password { get; set; }

    public virtual ICollection<Message> SendedMessages { get; set; }

    public virtual ICollection<Message> ReceivedMessages { get; set; }
}

public class Message
{
    public virtual int MessageId { get; set; }
    public virtual string Subject { get; set; }
    public virtual string Content { get; set; }

    public virtual User SentBy { get; set; }
    public virtual User ReceivedBy { get; set; }
}
```

Con esta definición, a pesar de que aparentemente es correcta, veremos que el modelo relacional que se crea, y por lo tanto el candidato con el que podría trabajar, sería algo como lo presentado en la siguiente figura..

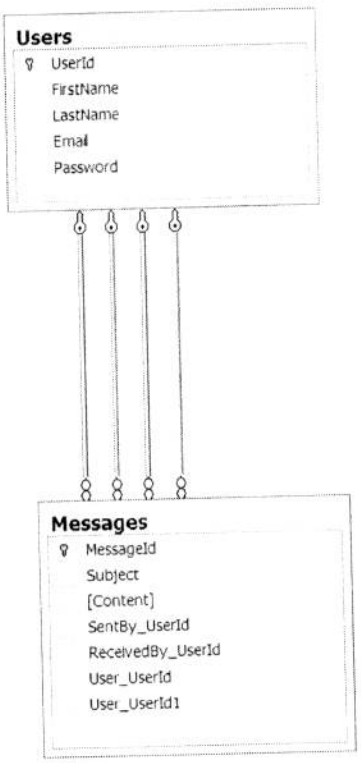


Figura 9.- Modelo de las asociaciones

¿Cuatro asociaciones? ¿Por qué?... La respuesta viene de dos hechos. El primero es porque no hemos establecido las claves externas, **ForeignKeyAttribute**, y por lo tanto las asociaciones son independientes (se crean las columnas para las mismas por convención de nombres como ya hemos explicado), y la segunda es que EF no ha podido interpretar las inversas de las asociaciones, y por lo tanto para *SentBy* y *ReceivedBy* vuelve a crearlas.

Para ir mejorando esto empecemos por tratar de indicar a EF cuáles son las inversas de las asociaciones. Para ello disponemos del atributo *InverseProperty*. Este atributo lo establecemos en la entidad *User* tal y como se ve a continuación.

```
public class User
{
    public virtual int UserId { get; set; }
    public virtual string FirstName { get; set; }
    public virtual string LastName { get; set; }
    public virtual string Email { get; set; }
    public virtual string Password { get; set; }
    [InverseProperty("SentBy")]
    public virtual ICollection<Message> SendedMessages { get; set; }

    [InverseProperty("ReceivedBy")]
    public virtual ICollection<Message> ReceivedMessages { get; set; }
}
```

Con este simple cambio, el modelo creado ya tendría un mejor aspecto, Figura 10.

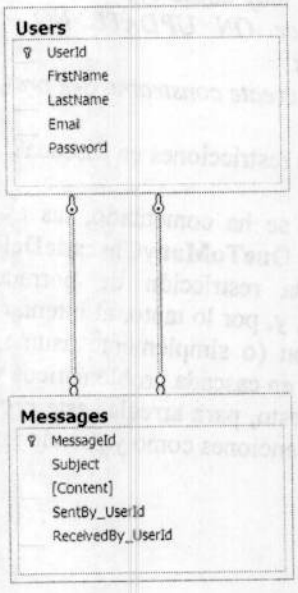


Figura 10.- Modelo de la asociación incluyendo InverseProperty

Ahora, solamente nos faltaría incluir nuestras propiedades de clave externa, como ya sabemos mediante el atributo **ForeignKey**:

```
public class Message
{
    public virtual int MessageId { get; set; }
    public virtual string Subject { get; set; }
    public virtual string Content { get; set; }

    [ForeignKey("SentBy")]
    public virtual int SentByUserId { get; set; }
    public virtual User SentBy { get; set; }

    [ForeignKey("ReceivedBy")]
    public virtual int ReceivedByUserId {get;set;}
    public virtual User ReceivedBy { get; set; }
}
```

Nota: Fíjese como en el caso de uso anterior el atributo *ForeignKey* se aplicó a la propiedad de navegación para indicar qué propiedad era la clave externa. Ahora, se ha aplicado a la propiedad de clave externa para indicar cuál es la navegación. Este atributo admite el uso de ambos casos como puede observar en la propia ayuda del mismo

Desde luego, ahora con este cambio, todo parece tener ya el aspecto adecuado, sin embargo, si intentáramos trabajar con este modelo, durante la creación de la base de datos, tendríamos una excepción como la siguiente:

"Introducing FOREIGN KEY constraint 'User_SendedMessages' on table 'Messages' may cause cycles or multiple cascade paths. Specify ON DELETE NO ACTION or ON UPDATE NO ACTION, or modify other FOREIGN KEY constraints.
Could not create constraint. See previous errors."

¿Ciclos de restricciones en cascada?...

Como ya se ha comentado, las asociaciones de clave externa disponen de una convención, **OneToManyCascadeDeleteConvention**, por la cual a éstas se les establece una restricción de borrado en cascada. En este caso tenemos dos asociaciones, y, por lo tanto, al intentar poner esta restricción en las dos, Sql Server se encuentra con (o simplemente asume que podría existir) un problema de *path* de restricciones en cascada problemáticos y lanza una excepción.

Por supuesto, para arreglar este problema, podríamos eliminar la convención de la lista de convenciones como ya se vio anteriormente.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions
      .Remove<OneToManyCascadeDeleteConvention>();
}
```

Una vez hecho, el diagrama de la base de datos ya es el esperado, Figura 11.

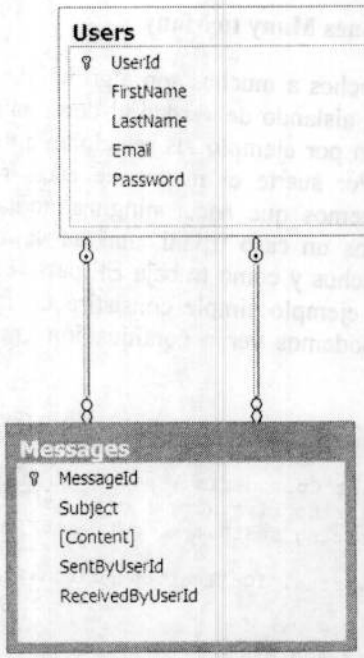


Figura 11.- Diagrama de la base de datos para las asociaciones

Bueno, por descontado, aunque aún no hemos visto nada para las relaciones one to many, este mismo trabajo podemos (debemos? :-)) realizarlo con nuestra API fluent. Si para las relaciones one to one teníamos métodos como *HasRequired* o *HasOptional*, ahora, veremos como también disponemos del método *HasMany*, presentado en una tabla anterior. A continuación, en el siguiente fragmento de código, se muestra una posibilidad de mapeo que resuelve este mismo modelo sin necesidad de anotar ninguna propiedad de las entidades.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   modelBuilder.Entity<User>()
               .HasMany(c => c.SendedMessages)
               .WithRequired(m => m.SentBy)
               .HasForeignKey(m => m.SentByUserId)
               .WillCascadeOnDelete(false);

   modelBuilder.Entity<User>()
```

```
                .HasMany(c => c.ReceivedMessages)
                .WithRequired(m => m.ReceivedBy)
                .HasForeignKey(m => m.ReceivedByUserId)
                .WillCascadeOnDelete(false);
    }
```

Fíjese como, a mayores del método **HasMany** utilizamos los métodos **WithRequired**, **HasForeignKey** y **WillCascadeOnDelete**. Aunque se entienden bien por el nombre, **WithRequired** nos permite indicar la inversa de la asociación por medio de una expresión lambda. Por otra parte, al igual que el atributo **ForeignKey**, con el método **HasForeignKey** podemos especificar cuál es la propiedad de clave externa de la asociación. Para finalizar, el tipo de restricción en cascada puede establecer con **WillCascadeOnDelete**.

3. Asociaciones Many to Many

Las relaciones muchos a muchos son algo habitual y realmente productivo en los modelos de dominio, aislando de verdad al desarrollador de elementos superfluos de cara a éste, como son por ejemplo las relaciones intermedias que se necesitan en un modelo relacional. Por suerte el mapeo de estas relaciones es sencillo, tanto que generalmente no tenemos que hacer ninguna anotación ni mapeo especial. Como siempre, expondremos un caso trivial, aunque válido, para ver la casuística de un mapeo muchos a muchos y como trabaja EF para realizar su mapeo/asignación a una base de datos. Este ejemplo simple consistirá en la típica relación entre clientes y direcciones, como podemos ver a continuación con nuestras entidades *Customer* y *Address*:

```
public class Customer
{
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public virtual ICollection<Address> Addresses { get; set; }

}
public class Address
{
    public int AddressId { get; set; }
    public string City { get; set; }
    public string ZipCode { get; set; }
    public string Street { get; set; }

    public virtual ICollection<Customer> Customers { get; set; }
}
```

Sin más, simplemente definiendo nuestras navegaciones, por medio de *ICollection*, conseguimos en código el efecto de una asociación muchos a muchos.

Nota: Recuerde, en el capítulo sobre objetos POCO y la generación de *proxies* en ADO.NET Entity Framework, como los objetos *proxies*, para su correcta generación imponían a las propiedades de navegación de tipo colección estar definidas obligatoriamente con el tipo *ICollection* genérico.

Como siempre, si revisáramos el modelo relacional generado veríamos como éste estaría preparado para trabajar-crear el siguiente esquema.

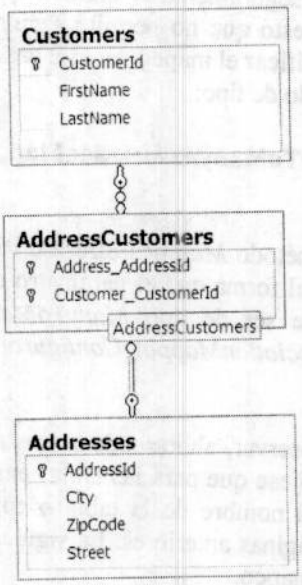


Figura 12.- Esquema relacional de la asociación muchos a muchos

Si ha sido observador sabrá que durante estos párrafos se he dicho cómo este mapeo nos permitirá crear un esquema para el trabajo, o bien como siempre, adaptar el modelo de entidades a una base de datos existente. Sin embargo, fíjese como en este caso los nombres de las columnas de la tabla intermedia se han creado por convención.

Nota: La convención sobre la configuración como ya habrá observado es un elemento habitual en EF 4.1 y Code First. Realmente, este no es un concepto nuevo, ya está implementado en otras muchas tecnologías, .NET y no .NET. Puede ver más información sobre esto en:

http://msdn.microsoft.com/en-us/magazine/dd419655.aspx

Si estamos ante este caso no se preocupe, porque esta especificación podremos realizarla de una forma sencilla con nuestro fluent API. En el siguiente fragmento de código puede ver el método de creación del modelo de la unidad de trabajo:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   modelBuilder.Entity<Customer>()
               .HasMany(c => c.Addresses)
               .WithMany(c => c.Customers)
               .Map(c
                    =>
                    {
                        c.ToTable("CustAdd");
                        c.MapLeftKey("CID");
                        c.MapRightKey("AID");
                    });
}
```

De este código, seguramente el método que le llame más la atención será **Map**, método terminal puesto que no permite seguir la cadena de llamadas. Nos ofrece la posibilidad de especificar el mapeo, en este caso de una relación muchos a muchos, por medio de un delegado de tipo:

Action<ManyToManyAssociationMappingConfiguration>.

Nota: Este método *Map* nos ofrece distintas sobrecargas, en función de lo que mapeamos, de tal forma que el parámetro del delegado *Action* que tenemos como parámetro puede ser de tipo *ManyToManyAssociationMappingConfiguration* o *ForeignKeyAssociationMappingConfiguration*.

Como puede observar, ahora podemos establecer cuál tiene que ser la forma de la tabla intermedia. Fíjese que para las tablas extremas se está dando por hecho que sabe cómo configurar el nombre de la tabla o columnas, puesto que lo hemos visto con claridad en unas páginas anteriores. La siguiente figura muestra el esquema relacional obtenido con este mapeo.

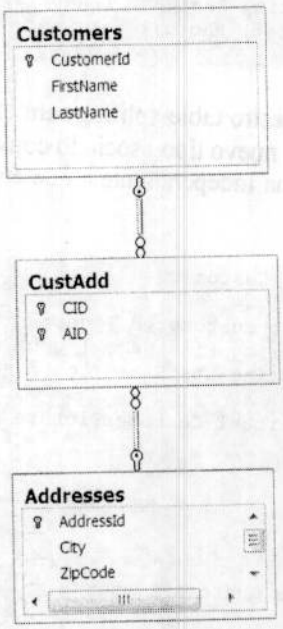


Figura 13.- Esquema relacional con el mapeo de la tabla intermedia

3.5.4.- Table Splitting

Si recuerda, en el capítulo 2 de este libro, en concreto en el punto 1.1.3, se definió el *Table Splitting* de la siguiente manera:

"..un ejemplo claro dónde la creación de un asociación nos aporta un gran valor dentro de la definición de un modelo de entidades es el uso de Table Splitting. Básicamente, Table Splitting, es el hecho de dividir una entidad de nuestro modelo de entidades en varias, dos o más, y realizar asociaciones uno a uno entre la entidad principal y las nuevas entidades creadas. Los razonamientos y elementos que motivan el uso de esta técnica son variados. Un motivo, por ejemplo, podría ser un deficiente diseño del modelo relacional, por el cual una tabla tuviera una ingente cantidad de columnas, y por lo tanto, la entidad creada dentro de EDM dispusiera, lógicamente, de una cantidad de propiedades equivalente. Otro de los ejemplos que nos podría hacer pensar en utilizar la técnica de Table Splitting sería la separación en un modelo de entidades de objetos valor, Value Objects y Entities por medio también de asociaciones uno a uno..."

Pues bien, tanto mediante el uso de atributos como mediante nuestra fluent API, este mismo escenario puede ser resuelto en Code First. Para verlo intentaremos realizar

el mismo ejemplo que el propuesto en el capítulo 2. Empezaremos poniéndonos en la situación de que tenemos la siguiente entidad *Customer*.

```
public class Customer
{
   public int CustomerId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public byte[] Photo { get; set; }
}
```

El objetivo para nuestro table-spliting será el de eliminar la propiedad *Photo* de esta entidad y llevarla a un nuevo tipo asociado con *Customer*, de tal forma que sea parte de este agregado, de forma independiente a que en la base de datos exista una única tabla para este escenario.

```
public class Customer
{
   public int CustomerId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }

   public virtual CustomerPicture Picture
   {
      get;
      set;
   }

}
public class CustomerPicture
{
   public int CustomerId { get; set; }
   public byte[] Photo { get; set; }

}
```

Seguramente, esto ya le recuerda a algo, ¿verdad?.

En realidad es un tipo complejo tal cual lo hemos explicado anteriormente. Pero ¿nos vale el concepto de tipo complejo para este caso?, es decir, dicho de otra forma: ¿es lo mismo un tipo complejo que un *table spliting*?.

Pues **no**, puesto que por definición un tipo complejo es la agrupación de una serie de propiedades dentro de una entidad en un tipo valor, pero este tipo es cargado siempre con la entidad. Al contrario que esto, con *table spliting* disponemos de una asociación que podremos cargar o no en nuestro modelo de dominio, pero no en el relacional, que será una sola tabla.

Nota: Recuerde aquí que el hecho de que EF interprete el fragmento anterior como un tipo complejo o una asociación depende de si es capaz de "ver" cual sería la clave de la asociación (en el caso presentado *CustomerPicture*). Como no existe ni propiedad *Id*, ni propiedad *CustomerPictureId*, no ha sido capaz y por lo tanto lo interpreta como un *ComplexType*. Si lo necesita, repase el punto dedicado a tipos complejos y asociaciones uno a uno.

A nuestro trabajo para hacer un *table spliting* tal y como queremos tendremos que configurar nuestro mapeo como se puede apreciar en el siguiente fragmento de código:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
                .ToTable("Customers");

    modelBuilder.Entity<CustomerPicture>()
                .HasKey(cp => cp.CustomerId)
                .ToTable("Customers");


    modelBuilder.Entity<Customer>()
                .HasRequired(c => c.Picture)
                .WithRequiredDependent();
}
```

Fíjese como se establece la misma tabla para ambas entidades, y puesto que EF, por convención, no es capaz de extraer cual es la clave de la entidad *CustomerPicture*, se realiza este trabajo imperativamente con el método **HasKey**. Una vez configuradas las entidades se procede a mapear la asociación

Si miráramos el modelo relacional asociado, veríamos que este mapeo es compatible con la siguiente tabla, figura 14.

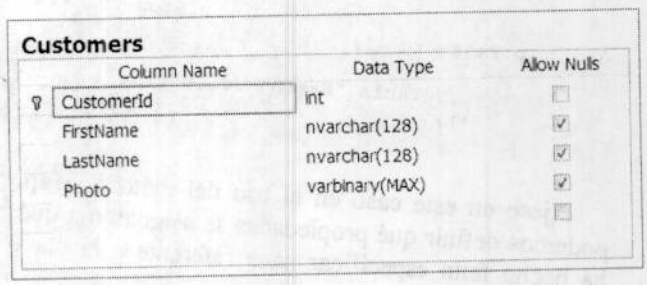


Figura 14.- Esquema asociado al Table Spliting del ejemplo

3.5.5.- Entity Splitting

Entity Splitting se podría resumir como el caso contrario al anterior, es decir, si antes teníamos una tabla que dividíamos en varias entidades, ahora nuestro objetivo es disponer de una entidad con ciertas propiedades mapeadas en una tabla y otras propiedades en otra tabla. Vamos a partir otra vez de un ejemplo que nos permita ver esto de una forma sencilla. Imaginemos que disponemos de una entidad *Usuario* con los atributos habituales y atributos referidos por ejemplo a su autenticación, algo como lo presentado en el siguiente fragmento.

```
public class User
{
   public int UserId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string Email { get; set; }
   public string UserName { get; set; }
   public string Password { get; set; }
}
```

Es probable que a pesar de hace que la información sobre la autenticación (nombre de usuario y contraseña) forme parte de nuestra entidad, en el modelo relacional ésta esté dividida en diferentes tablas, relacionadas (1..0,1) entre sí. Si tenemos este escenario y deseamos mapear esto como una única entidad o bien vamos a generar el esquema de esta base de datos como acabamos de comentar, tendríamos que realizar el mapeo de esta entidad tal y como sigue:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   modelBuilder.Entity<User>()
               .Map(u
                    =>
                    {
                        u.Properties(p => new { p.FirstName,
p.LastName, p.Email });

    u.ToTable("User");
     });
     modelBuilder.Entity<User>()
                 .Map(u =>
                      {
                          u.Properties(p => new { p.UserName,
 p.Password });

     u.ToTable("UserSecurity");
     });
 }
```

Fíjese en este caso en el uso del método **Map,** y como, gracias a este método, podemos definir qué propiedades se asignarán a qué tabla. En este caso note como no ha hecho falta especificar nada referente a la clave, puesto que por convención EF

automáticamente descubre nuestra intención y realiza la tarea tal y como deseamos. A continuación se puede ver la estructura del modelo relacional creado con este mapeo.

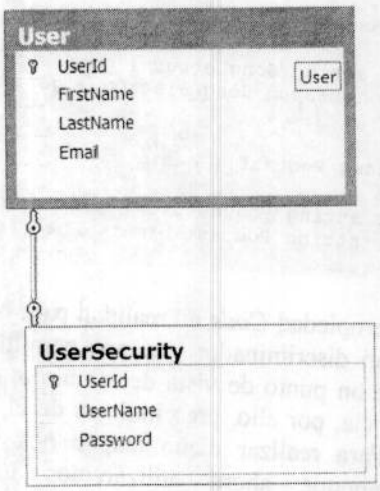


Figura 15.- Modelo relacional creado para el mapeo de entity splitting

3.5.6.- Herencia

En el capítulo 2, concretamente en el caso práctico 1.1.5, pudimos comprobar como Entity Data Model nos permitía "mapear" las distintas estrategias de herencia que podemos representar en un modelo relacional. A lo largo de este punto veremos esos mismos casos prácticos llevados a nuestro nuevo modelo de trabajo, sin necesidad de utilizar un fichero edmx, tal y como hemos hecho para los puntos anteriores.

1. Herencia por Jerarquía (TPH)

Con el fin de presentar para esta serie de casos los mismos ejemplos utilizados en el capítulo 2, partiremos para este caso de un entidad *Audio*, tal cual se puede ver en el siguiente fragmento de código:

```
public class Audio
{
    public int AudioId { get; set; }
    public string Code { get; set; }
    public string SongLetter { get; set; }
    public TimeSpan SongDuration { get; set; }
    public string PodcastUri { get; set; }
    public string PodcastFormat { get; set; }
}
```

Sobre esta entidad podríamos destacar realmente varios conceptos. Por un lado un *Song* y por otro un *Podcast*, de esta forma, si no quisiéramos sacrificar nuestro diseño de modelo de dominio, en realidad, en vez de escribir esta entidad tal y como está haríamos algo similar a lo siguiente:

```
public class Audio
{
   public int AudioId { get; set; }
}

public class Song : Audio
{
   public string SongLetter { get; set; }
   public TimeSpan SongDuration { get; set; }
}

public class Podcast : Audio
{
   public string PodcastUri { get; set; }
   public string PodcastFormat { get; set; }
}
```

Fíjese que la propiedad *Code* en realidad para nosotros en este ejemplo no actuaba más que como un discriminador que nos permitía discernir entre los dos tipos de elementos. Desde un punto de vista del dominio, en realidad, este elemento no tiene ninguna importancia, por ello, prescindimos de él, aunque en realidad, "tendrá efecto en el mapeo". Para realizar dicho mapeo nos valdremos del método **Map** visto anteriormente, aunque ahora utilizaremos un nuevo elemento de nuestro *EntityMappingConfiguration*, concretamente el método **Requires**:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Audio>()
                .Map(emc =>
                {
                        emc.Requires("Code")
                           .HasValue(null);

                })
                .Map<Song>(emc =>
                {
                        emc.Requires("Code")
                           .HasValue("song");


                })
                .Map<Podcast>(emc =>
                {
                        emc.Requires("Code")
                           .HasValue("podcast");


                });
}
```

Parémonos un poco a explicar este mapeo. En primer lugar vamos a configurar nuestra entidad *Audio*. Realmente, como ya sabe, cuando hay una herencia solamente

tenemos un "agregado de partida", que en **EDM** llamábamos **EntitySet**. Para esta configuración queremos establecer que la entidad requiere un elemento llamado *Code* cuyo valor sea nulo. Una vez hecho esto establecemos lo mismo para los distintos valores que pueda tener el discriminador. En realidad lo que acabamos de hacer es lo siguiente: la entidad *Audio* tiene un discriminador llamado *Code* como nulo, la entidad *Song* con el valor *song* y la entidad *Podcast* con el valor *podcast*. Con esto, si viéramos nuestro modelo relacional, en realidad una tabla como era nuestro propósito, veríamos que tendremos lo siguiente:

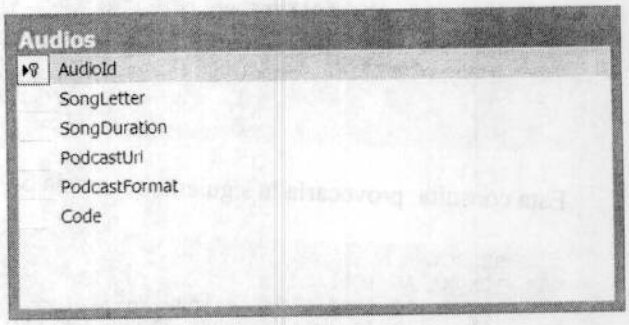


Figura 16.- Esquema del modelo para TPH

Ahora, una vez mapeado, podríamos probar por ejemplo la escritura de instancias de estos tres tipos, como se ve a continuación:

```
using (ZuneUnitOfWork unitOfWork = new ZuneUnitOfWork())
{
    Audio audio = new Audio(){};

    Song song = new Song()
    {
       SongDuration = new TimeSpan(0,4,21),
       SongLetter = "Dolores se llamaba Lola.."
    };

    Podcast podcast = new Podcast()
    {
       PodcastFormat = "xx",
       PodcastUri = "http://www.plainconcepts.com/podcast/"
    };


    unitOfWork.Audios.Add(audio);
    unitOfWork.Audios.Add(song);
    unitOfWork.Audios.Add(podcast);

    unitOfWork.SaveChanges();
}
```

Una vez que se han guardado estos elementos en la tabla **Audios,** podríamos observar como automáticamente, EF, en cada inserción ha asignado el valor correcto de

nuestro discriminador. La siguiente figura muestra los resultados de una consulta sobre esta tabla dentro de **Sql Server Management Studio**:

Results | Messages

	AudioId	SongLetter	SongDuration	PodcastUri	PodcastFor	Code
1	1	NULL	NULL	NULL	NULL	NULL
2	2	Dolores se llamaba Lola, hace la calle	00:04:21.0000000	NULL	NULL	song
3	3	NULL	NULL	http://www.plainconcepts.com/podcast/	xx	podcast

Figura 17.- Datos después de la inserción

Por supuesto, las consultas sobre los distintos tipos de la jerarquía harán uso también de estos valores asignados al discriminador. Un ejemplo de esto es la siguiente consulta Linq que nos permite obtener los elementos de tipo *Song*.

```
using (ZuneUnitOfWork unitOfWork = new ZuneUnitOfWork())
{
      var result = unitOfWork.Audios
                             .OfType<Song>()
                             .ToList();

}
```

Esta consulta, provocaría la siguiente sentencia SQL en nuestra base de datos:

```
SELECT
'0X0X' AS [C1],
[Extent1].[AudioId] AS [AudioId],
[Extent1].[SongLetter] AS [SongLetter],
[Extent1].[SongDuration] AS [SongDuration]
FROM [dbo].[Audios] AS [Extent1]
WHERE [Extent1].[Code] = 'song'
```

Nota: Esta opción de herencia TPH es la opción de herencia por defecto en ADO.NET Entity Framework 4.1. Es decir, si no incluyéramos nuestro código de mapeo, bien mediante un *EntityTypeConfiguration* o directamente en el model builder automáticamente se seleccionaría esta estrategia. Como convención EF crearía en la tabla una columna llamada **Discriminator** haciendo la tarea de discriminador. Recuerde otra vez aquí el principio de **Convention over Configuration**.

2. Herencia Tabla por Tipo (TPT)

En ocasiones el modelo relacional anterior puede que no sea el que esté disponible o bien no nos guste. Podrían achacársele problemas como el crecimiento de página, la gran cantidad de elementos nulos para representar los distintos tipos de entidades, etc.

Como comentamos en el capítulo sobre mapeo con **EDM**, es posible que nos queramos decantar por una estrategia como **TPT**. Recuerde aquí que con la estrategia

TPT existiría una tabla para los datos comunes, nuestra entidad base *Audio*, y tablas para cada una de las subclases de ésta, *Song* y *Podcast*. Implementar esta estrategia es sumamente sencillo. Basta con indicar a cada tipo que se dirija a una tabla en particular, con el método ya conocido **ToTable**. A continuación, en el siguiente fragmento puede ver un ejemplo de esto:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Audio>()
                .ToTable("Audios");

    modelBuilder.Entity<Song>()
                .ToTable("Songs");

    modelBuilder.Entity<Podcast>()
                .ToTable("Podcasts");
}
```

Con esta sencilla configuración, ahora, nuestro modelo relacional de trabajo sería como el siguiente:

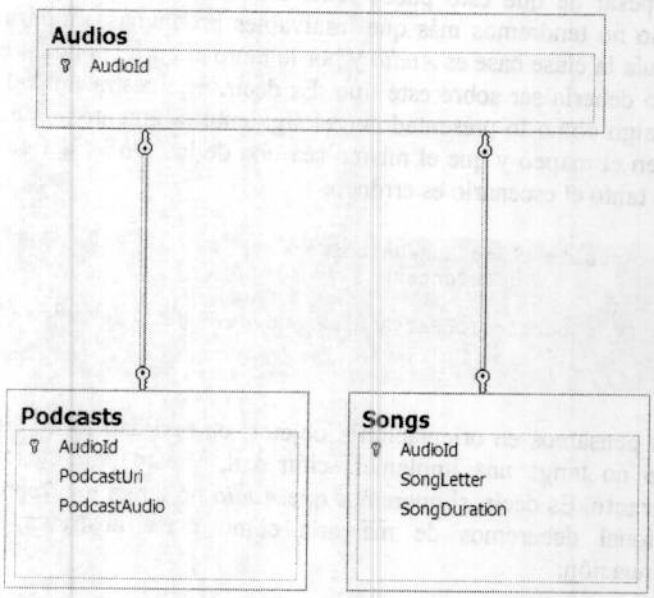


Figura 18.- Modelo relacional para TPT

Nota: Aunque no se está comentando en los ejemplos anteriores, usted puede configurar cualquier otro elemento de las propiedades de la entidad: nombre de las columnas, estrategias de generación, tamaños, requeridas o no etc.

Por supuesto si quisiera hacer este mapeo mediante anotaciones sería tan simple como establecer el atributo *TableAttribute* a nuestras entidades indicando el nombre de la tabla correspondiente.

3. Herencia Tabla por Tipo Concreto (TPC)

Como ya sabrá esta es la tercera de las estrategias que podemos usar para simular la herencia en modelos relacionales. Aunque en el capítulo 2 se hizo un ejemplo de la misma se comentó cómo esta estrategia no estaba soportada por el diseñador, y como esto podría hacernos la vida mucho más difícil puesto que trabajar directamente sobre el fichero XML de nuestro EDMX no es algo recomendable. Por suerte, en Code First no tenemos esa problemática: no tenemos edmx y por lo tanto esta estrategia será totalmente soportada.

La implementación en el modelo relacional como sabe es la de disponer de una tabla con los elementos comunes y particulares de cada uno de los elementos de la jerarquía. Revise el capítulo 2 si quiere obtener más información sobre los argumentos para seleccionar una u otras estrategias.

La primera intuición para realizar este mapeo suele basarse en la de eliminar del mismo la entidad *Audio*, puesto que, como acabamos de comentar, no existirá como tabla. Para esto se podría recurrir al método **Ignore**, tal cual se ve a continuación:

```
modelBuilder.Ignore<Audio>();
```

A pesar de que esto puede tener todo el sentido, en realidad, partiendo de este proceso no tendremos más que insalvables problemas, empezando porque en nuestra jerarquía la clase base es *Audio* y por lo tanto el correspondiente *DbSet* de la unidad de trabajo debería ser sobre este tipo. Es decir, en nuestra unidad de trabajo deberíamos tener algo como lo presentado en el siguiente fragmento. En esta situación ignorar un tipo en el mapeo y que el mismo sea uno de los *DbSet* del contexto es incompatible, por lo tanto el escenario es erróneo:

```
public class ZuneUnitOfWork
        : DbContext
{
    public IDbSet<Audio> Audios { get; set; }
}
```

Si pensamos en orientación a objetos, en realidad lo que queremos es que el tipo *Audio* no tenga una implementación real, y esto para nosotros es el concepto de **Abstracto**. Es decir, si queremos que *Audio* no tenga una representación en el modelo relacional deberemos de marcarla como clase abstracta, tal y como vemos a continuación:

```
public abstract class Audio
{
        public int AudioId { get; set; }
```

```
}
```

Ahora, una vez hecho esto, solamente nos quedaría indicar que cuando se realice el mapeo de nuestras entidades *Song* y *Podcast* también se incluyan las propiedades heredadas de la base, *Audio*, las cuales, recordemos no tendrán implementación en *Audio*. Para hacer esto, el elemento **EntityMappingConfiguration** de nuestro método **Map** nos ofrecerá el método **MapInheritedProperties**.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   modelBuilder.Entity<Song>()
               .Map(mc =>
               {
                   mc.MapInheritedProperties();
               })
               .ToTable("Songs");

    modelBuilder.Entity<Podcast>()
                .Map(mc =>
                {
                   mc.MapInheritedProperties();
                })
                .ToTable("Podcasts");
}
```

A continuación, puede ver el resultado de este mapeo en nuestro modelo relacional.

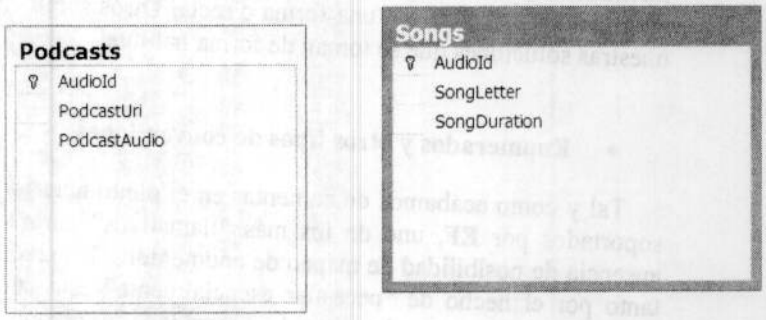


Figura 19.- Modelo relacional para TPC

3.5.7.- Elementos no soportados

Para nuestra pena, algunos elementos que hubiéramos considerado deseables y otros esenciales no están soportados en EF 4.1, tanto desde el punto de vista de Code First como para ciertos elementos de las nuevas API que presentaremos más adelante. De entre los elementos más comentados por la comunidad se encuentra el soporte para

enumerados y tipos espaciales, aunque podremos ver una posible solución alternativa en el siguiente punto.

En realidad, más que el soporte para enumerados y otros tipos, podríamos decir que no está soportada la posibilidad de hacer conversiones de tipos de valores como se puede hacer por ejemplo en NHibernate con sus *IValueType*. A mayores del soporte para esta casuística otra nota llamativa es la ausencia de soporte para **procedimientos almacenados**, algo que si tenemos disponible en nuestras API de *ObjectContext*.

Para terminar con la lista de elementos no soportados podremos destacar algo que habría sido de gran utilidad, como es el soporte para la definición de **convenciones personalizadas**, que sí hemos tenido durante las distintas CTP de EF 4.1, pero que por temas de calidad se quedó fuera de la *release* actual.

Para terminar con el último punto de los elementos no soportados tenemos que destacar la ausencia de **soporte de migración** o mejor dicho del soporte para cambios en la definición del modelo y de la adaptación de la base de datos. Puede ver más acerca de este trabajo y su posible introducción en una versión futura de EF en el siguiente enlace del blog EF Design:

http://blogs.msdn.com/b/efdesign/archive/2010/10/22/code-first-database-evolution-aka-migrations.aspx

3.5.8.- En la vida real...

En el capítulo 5 de este libro introdujimos al lector en muchas de las consideraciones de uso de un ORM en la vida real, lógicamente particularizándolo en Entity Framework, incluyendo los patrones más habituales en modelos de dominio como aggregates, repository, unit of work etc...

A lo largo de este epígrafe trataremos de ver algunos elementos corrientes en nuestro trabajo con EF 4.1 y Code First que tendremos que tratar de resolver para facilitarnos la vida. Algunos de estos puntos serán *workarounds* para necesidades que no podemos satisfacer de una forma directa. Otros serán consideraciones de diseño de nuestras soluciones que se toman de forma habitual.

- **Enumerados y otros tipos de conversiones**

Tal y como acabamos de comentar en el punto anterior acerca de los elementos no soportados por **EF**, uno de los más "llamativos" para la comunidad consiste en la ausencia de posibilidad de mapeo de enumerados. En realidad, esta "protesta" no viene tanto por el hecho de "necesitar esencialmente" esta característica del lenguaje que algunos denostamos, sino por la necesidad de disponer de algún mecanismo de extensibilidad de EF que nos permitiera adaptar el tipo recuperado de la base de datos a un nuevo tipo, y viceversa por supuesto, tal y como nos permiten otros ORM actualmente.

Por supuesto, como para casi todas las cosas, para esto podemos proponer algún tipo de solución alternativa. Supongamos que tenemos nuestra ya habitual definición de una entidad *Customer*.

```
public class Customer
{
   public int CustomerId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string Type { get; set; }
}
```

En esta entidad, como puede observar, disponemos de una propiedad *Type* que podría ser un posible enumerado (lógicamente se está forzando el modelo para explicar esta casuística). Con esto, decidimos definir este enumerado como sigue:

```
public enum CustomerType
{
    None=0,
    Premium = 1,
    Gold = 2
}
```

Ahora, podríamos incluir una nueva propiedad en nuestra entidad tal y como se puede ver en el siguiente fragmento de código:

```
public class Customer
{
   public int CustomerId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string Type { get; set; }

   public CustomerType CustomerType
   {
      get
      {
         CustomerType result;

         if (Enum.TryParse<CustomerType>(Type, out result))
             return result;
         else
             return CustomerType.None;
      }
      set
      {
        Type = Enum.GetName(typeof(CustomerType), value);
      }
   }
}
```

De esta forma, en la creación de una nueva entidad podríamos hacer algo tal que así:

```
Customer newCustomer = new Customer()
{
   FirstName = "Unai",
   LastName = "Zorrilla",
   CustomerType = CustomerType.Gold
};
```

Guardándose en la base de datos el valor correcto para nuestra propiedad *Type*. Por supuesto después de ver este *workaround* alguno "chillará" puesto que la propiedad *Type* sigue siendo pública y por lo tanto hay una dualidad en cuanto a como se establece o se puede establecer el valor del tipo del cliente, dejando una puerta abierta a que se establezca un posible valor de *Type* que no esté representado *realmente* en el enumerado. Realmente esto es así. Si exponemos la propiedad *CustomerType* no deberíamos exponer nuestra propiedad *Type*, por la razón ya comentada. Para ello, tendremos que buscar una manera de realizar esta tarea, o mejor dicho, buscar una manera de soportar algo que por defecto no tenemos y es el **mapeo de miembros privados**.

Como ya hemos visto en muchas ocasiones en el apartado sobre mapeo, nuestro API fluent nos ofrece (para las entidades o tipos complejos) un método llamado **Property**, con sus distintas sobrecargas para los distintos tipos primitivos. Éste nos permite indicar el mapeo de una propiedad primitiva. Este método **Property** esta definido en **StructuralTypeConfiguration<TEntity>.** A continuación se muestra un ejemplo de una de las sobrecargas de este método el cual, básicamente, nos permite indicar mediante una expresión lambda que propiedad deseamos configurar de nuestra entidad o tipo complejo.

```
public abstract class StructuralTypeConfiguration<TStructuralType>
where TStructuralType : class
{
    // Omitido por brevedad

    public PrimitivePropertyConfiguration
Property<T>(Expression<Func<TStructuralType, T>> propertyExpression)
where T : struct;

    // omitido por brevedad

}
```

Por desgracia con esta firma no podemos indicar una propiedad privada, ya que la expresión **Expression<Func<TStructuralType,T>>** lógicamente no nos lo permite.

Lo que necesitaríamos sería una forma de obtener un elemento de este tipo anterior sin necesidad de que *T* fuera una propiedad pública. Para ello podríamos utilizar las capacidades de los árboles de expresión y hacer uso de las mismas para formar nuestro propio árbol y usarlo posteriormente.

A continuación, puede ver una de las distintas sobrecargas del método *Property* con soporte para miembros privados. Este código y el del resto de sobrecargas, acompaña al código de ejemplo del libro en un proyecto llamado **EFExtensions**.

```
/// <summary>
/// Extension method for map private properties
/// <example>
/// modelBuilder.Entity<Customer>()
///             .Property<Customer,int>("Age")
///             .IsOptional()
/// </example>
/// </summary>
/// <typeparam name="TEntityType">The type of ///entity to
map</typeparam>
/// <typeparam name="KProperty">The type of ///private property to
map</typeparam>
/// <param name="entityConfiguration">Asociated
///EntityTypeConfiguration</param>
///
///<param name="propertyName">The name of private ///property
///</param>
/// <returns>A PrimitivePropertyConfiguration for /// this map
///</returns>
public static PrimitivePropertyConfiguration Property<TEntityType,
KProperty>(this EntityTypeConfiguration<TEntityType>
entityConfiguration, string propertyName)
            where TEntityType : class
            where KProperty:struct
{

   var propertyInfo = typeof(TEntityType).GetProperty(propertyName,
BindingFlags.NonPublic

|

BindingFlags.Instance

|

BindingFlags.Public);
    if ( propertyInfo != null ) // if private property exists
    {
        ParameterExpression arg =
Expression.Parameter(typeof(TEntityType), "parameterName");
        MemberExpression memberExpression =
Expression.Property((Expression)arg, propertyInfo);

         //Create the expression to map
         Expression<Func<TEntityType, KProperty>> expression =
(Expression<Func<TEntityType,
KProperty>>)Expression.Lambda(memberExpression, arg);


          return entityConfiguration.Property(expression);
    }
    else
        throw new InvalidOperationException("The property does not
exist");
}
```

```
var propertyInfo = typeof(TEntityType).GetProperty(propertyName,
BindingFlags.NonPublic | BindingFlags.Instance
|
BindingFlags.Public);
```

```
if ( propertyInfo != null )
{
     ParameterExpression arg =
Expression.Parameter(typeof(TEntityType), "parameterName");
     MemberExpression memberExpression =
Expression.Property((Expression)arg, propertyInfo);

     //Create the expression to map
     Expression<Func<TEntityType, KProperty>> expression =
(Expression<Func<TEntityType,
KProperty>>)Expression.Lambda(memberExpression, arg);


               return entityConfiguration.Property(expression);
          }
```

Una vez que disponemos de esta nueva capacidad de mapeo de miembros privados podemos realizar el mapeo tal y como tenemos anteriormente, pero ahora dejando nuestra propiedad *Type* sin acceso fuera de la clase y, por lo tanto, tal y como estábamos buscando.

A continuación, en los siguientes fragmentos, podemos ver el uso de nuestros métodos extensores y la definición de la entidad Customer.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   modelBuilder.Entity<Customer>()
               .PropertyString<Customer>("Type");
}
```

Lógicamente, este "pequeño truco" utilizado para el uso de enumerados también es válido para el trabajo con otros tipos de datos como por ejemplo podrían ser los datos espaciales de **Sql Server.** Como seguramente sabe este tipo de datos aún no están soportados por Entity Framework, ni siquiera en esta nueva versión. Por lo tanto no podemos disponer de una entidad mapeada a una tabla con una columna espacial (*geography*, *geometry*, etc.).

Un posible *workaround* para esto es hacer el mapeo de nuestras columnas *geography* como *blobs*, puesto que hay una conversión explícita posible de los tipos de datos geográficos a este tipo de datos. Así por ejemplo, podríamos definir una entidad como sigue:

```
public class ProductLocation
{
    public int Id { get; set; }

    byte[] BinaryLocation;
    public SqlGeography Location
    {
       get
       {
         return SqlGeography.STGeomFromWKB(new
SqlBytes(BinaryLocation), 4236);
       }
       set
```

```
            {
                BinaryLocation = (value.STAsBinary()).Buffer;
            }
        }
    }
```

Fíjese que en esta entidad, se utiliza el mismo "truco" que con los enumerados: definir una propiedad privada -que será la que realmente almacenemos en nuestra base de datos-, y una propiedad pública -que utilizaremos para leer y escribir de una forma más adecuada pero sin estar mapeada a una tabla de la base de datos (como hemos dicho por no estar soportado por ahora este tipo de datos).

Nota: Para utilizar SqlGeography así como el resto de tipos espaciales es necesario incorporar una referencia a *Microsoft.SqlServer.Types.dll*.

- **Agregando comportamiento a nuestras entidades**

De forma general, uno de los principales defectos en la creación de modelos de dominio (sobre todo para aquellos que están empezando en este mundo) es el del antipatrón conocido como "anemic domain model" (véase Anemic Domain Model, Martin Fowler, **http://www.martinfowler.com/bliki/AnemicDomainModel.html**).

Este antipatrón hace referencia al hecho de disponer de un modelo de entidades que únicamente resultan en una bolsa de datos, es decir, entidades con solamente *geters* y *seters* que no disponen de ningún tipo de comportamiento. Por experiencia, cuando se describe este hecho a personas que están creando sus primeros modelos de dominio, generalmente la gente empieza a ser reacia hablando sobre la inexistencia de comportamiento en muchas entidades. Poco a poco revisando el negocio y hablando sobre el concepto y propósito de cada una de las entidades (recuerde en la primera parte de este libro las diferencias entre *aggregate root*, entidades y *value objects*) uno se va dando cuenta de que este comportamiento existe. En ocasiones son pequeños métodos que nos validan el acceso y uso de ciertas propiedades. En otras ocasiones son una forma de envolver negocio de las entidades (por ejemplo, podría ser aquel que nos indica cuándo un cliente está bloqueado y cuándo no, de forma que este concepto no se encuentre desperdigado por distintos elementos de la solución). Significa en definitiva, dotar a las entidades del "sentido de la orientación a objetos", como dice Martin Fowler.

Muchos de estos comportamientos que agregamos a nuestras entidades implican cambios en nuestros mapeos, bien porque algunas propiedades dejan de ser públicas, otras desaparecen, otras se encapsulan etc. A continuación, pondremos un ejemplo muy simple con el fin de mostrar un posible comportamiento de entidades. Lógicamente para nuestro ejemplo omitiremos muchos elementos de negocio y otros elementos que deberían existir pero que, para nuestro propósito, no son necesarios.

Partiremos, para el ejemplo mencionado, de dos posibles entidades *Order* y *OrderLine* como se ve a continuación:

```
public class Order
{
    public int Id { get; set; }

    public DateTime OrderDate { get; set; }

    public virtual ICollection<OrderLine> Lines { get; set; }
}
```

```
public class OrderLine
{
    public int Id { get; set; }

    public decimal TotaLine { get; set; }

    public virtual Order Order { get; set; }

    public int OrderID { get; set; }
}
```

Estas entidades, tal y como están presentadas, serían la tendencia de todos aquellos que empiezan a jugar con modelos de dominio. Como observará, estas entidades son meros transmisores de datos (concepto que ya conocemos como DTO's). Hasta ahora, con EF 4.1, el trabajo con estas entidades era complicado puesto que las mismas son generalmente creadas automáticamente por alguna de las plantillas T4 que tenemos a nuestra disposición y, por lo tanto, acciones como la eliminación de navegaciones o la ocultación de propiedades era complicada y en muchas ocasiones imposible. Ahora, puesto que tenemos total control sobre el código, éste ya no se genera a patir de un EDM, sino que nosotros empezamos a codificar el modelo, podemos realizar muchas de las tareas mencionadas anteriormente.

Nota: En el siguiente punto veremos como también disponemos, con la instalación de EF 4.1, de nuevas plantillas de EDM para generar el código en base a DbSet y DbContext.

Uno de los pensamientos que podríamos tener (y se suele realizar en no pocas ocasiones) es ocultar de forma pública el acceso a alguna navegación de un agregado, en este caso a la colección de líneas de pedido, de tal forma que la eliminación o la agregación de una nueva línea de pedido pasaran por algún método del agregado, en este caso un método *AddLine*.

```
public class Order
{
    // ... Omitted for brevity

    ICollection<OrderLine> Lines
    {
        get
        {
```

```
            if (_lines == null)
                _lines = new HashSet<OrderLine>();

            return _lines;
        }
        set
        {
            Lines = value;
        }
    }


    public void AddLine(OrderLine line)
    {
        if (line != null)
        {
            //TODO: Add logic validation and
            //other things here!

            //add order line, preserve graph..
            line.Order = this;
            Lines.Add(line);
        }
    }
}
```

Lógicamente, **aquí no se termina el trabajo, en realidad no ha hecho nada más que comenzar**, hay otros muchos pensamientos que podríamos o tendríamos que agregar a nuestras entidades. En este caso podrían ser el cálculo de totales del pedido, sobre todo si esto implica lógica, tener en cuenta el estado, el cliente asociado, forma de cálculo de descuentos etc. Ejemplos más completos sobre este trabajo y las distintas técnicas que podemos utilizar se podrán encontrar próximamente, esperemos que en la publicación de este apéndice en el proyecto Microsoft NLayer App: **http://microsoftnlayerapp.codeplex.com** .

4.- NUEVAS API

Hasta este momento, de esta nueva versión de Entity Framework solamente nos hemos quedado con lo que llamamos Code First, o, las capacidades de pensar primero en el código de nuestro modelo, para después, por medio de convenciones y de un API fluent pasar a realizar/establecer los mapeos para trabajar con una nueva base de datos o una base de datos existente. Pero para esto hemos pasado de puntillas sobre un conjunto nuevo de APIs cuyo uso no solamente está circunscrito a Code First sino que podríamos hacer uso de las mismas con Model First o partiendo de la base de datos como hacemos desde la primera versión.

De hecho, si descargamos EF 4.1, en vez de hacer uso de sus capacidades con el paquete de NuGet, podremos ver como a mayores de los artefactos de generación que tengamos instalados, **EntityObject Generator**, **STE Generator** o **POCO Generator**, ahora tendremos uno nuevo con el nombre **"ADO.NET DbContext Generator"**, plantilla que por defecto se instala dentro del directorio de Visual Studio 2010.

Nota: Puede obtener el instalador de EF 4.1 RTW en:

http://www.microsoft.com/downloads/en/details.aspx?FamilyID=b41c728e-9b4f-4331-a1a8-537d16c6acdf&displaylang=en

De esta manera, con la forma de trabajo tradicional también podríamos hacer uso de las nuevas API **DbContext** y **DbSet** presentadas de forma somera en los puntos anteriores y que ahora ampliaremos.

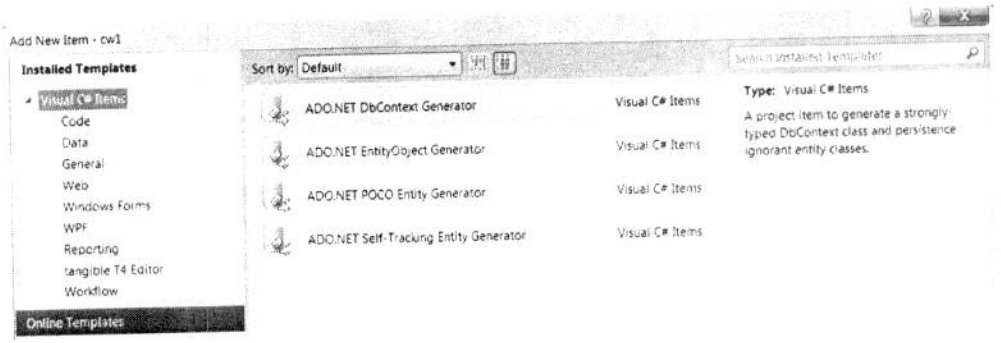


Figura 20.- Artefacto de generación DbContext Generator

Lógicamente, el aspecto de los elementos es idéntico al que nosotros usamos a lo largo de todo este capítulo, con la diferencia principal que ahora se sobrescribe el método de creación del modelo y en el mismo se lanza una excepción de tipo **UnintentionalCodeFirstException** . Esta excepción tiene como propósito notificar al desarrollador que se está haciendo uso de las API **DbContext** y **DbSet** partiendo de un modelo y que, por lo tanto, el proveedor de la cadena de conexión debe ser **System.Data.EntityClient** con la especificación por lo tanto de los diferentes espacios **SSDL**, **CSDL** y **MSL** a partir de los que se extraerá la información de mapeo como siempre hemos hecho.

```
public partial class NLayerAppEntities : DbContext
{
    public NLayerAppEntities()
    : base("name=NLayerAppEntities")
    {
    }

    protected override void OnModelCreating(DbModelBuilder
modelBuilder)
    {
      throw new UnintentionalCodeFirstException();
    }

    public DbSet<Country> Countries { get; set; }
    public DbSet<Customer> Customers { get; set; }
}
```

4.1.- DbContext nuestra nueva base

DbContext podría definirse como una versión ligera de **ObjectContext** que nos facilita el trabajo con los distintos elementos y conceptos que tenemos hoy en día en la base de nuestras unidades de trabajo. De hecho, la primera cosa que deberemos tener clara es que DbContext no es más que una *Adapter* de *ObjectContext* declarado por la interface **IObjectContextAdapter**.

```
public class DbContext : IDisposable, IObjectContextAdapter
{
   //Omitted for brevity
}
public interface IObjectContextAdapter
{
   ObjectContext ObjectContext { get; }
}
```

De entre los elementos más importantes de *DbContext* podríamos destacar su fácil y rápido acceso a la configuración del mismo y al trabajo con la base de datos subyacente por medio de sus propiedades **Configuration** y **Database**. La primera, nos permitirá establecer las siguientes características de trabajo de nuestros contextos.

Tabla 9.- Haga clic aquí para escribir texto.

Propiedad	Descripción
AutoDetecChangesEnabled	Permite establecer si la unidad de trabajo detectará cambios en las entidades bajo ciertas condiciones.
LazyLoadingEnabled	Permite establecer si la carga perezosa está habilitada, el valor por defecto es true. Lógicamente para que esta caraterística tenga efecto la generación de proxies tiene que está habilitada.
ProxyCreationEnabled	Permite establecer si la generación de objectos proxy para nuestras entidades está habilitada o no, el valor por defecto es true.
ValidateOnSaveEnabled	Permite habilitar la validación de las entidades en el proceso de comprometer los cambios efectuados.

Si *Configuration* nos permite un acceso rápido a ciertos *flags* de trabajo con nuestros contextos, la propiedad *Database* nos permitirá acceder a un elemento con capacidades para descubrir, validar y/o crear la base de datos subyacente a nuestro

modelo de entidades. Este elemento, visto anteriormente ya, se llama *Database* y la firma de sus principales características puede verse a continuación en el siguiente fragmento:

```
public class Database
{
   public DbConnection Connection { get; }

   public static IDbConnectionFactory DefaultConnectionFactory { get;
set; }

   public bool CompatibleWithModel(bool throwIfNoMetadata);

   public void Create();

   public bool CreateIfNotExists();

   public bool Delete();

   public static bool Delete(DbConnection existingConnection);

   public static bool Delete(string nameOrConnectionString);

   public int ExecuteSqlCommand(string sql, params object[]
parameters);

  public bool Exists();

  public static bool Exists(DbConnection existingConnection);

  public static bool Exists(string nameOrConnectionString);

  public void Initialize(bool force);

  public static void
SetInitializer<TContext>(IDatabaseInitializer<TContext> strategy)
where TContext : DbContext;

  public IEnumerable<TElement> SqlQuery<TElement>(string sql, params
object[] parameters);
}
```

Como habrá podido observar, gracias a este tipo, desde el propio contexto de trabajo podremos comprobar si la base de datos es compatible con el modelo, revisar su existencia, inicializarla o crearla, amén lógicamente de realizar consultas o comandos de una forma directa por medio de los métodos **SqlQuery** y **ExecuteSqlCommand.**

A mayores de los elementos anteriores, solamente por esto no hubiera merecido la pena el adaptador. Tenemos nuevas facilidades para todo lo que tiene que ver con el mantenimiento del estado de los distintos elementos en una unidad de trabajo, es decir, una simplificación de nuestro ya conocido **ObjectStateManager.** Esta simplificación, viene dada por los elementos presentados a continuación:

```
public class DbContext : IDisposable, IObjectContextAdapter
{
 public DbChangeTracker ChangeTracker { get; }
```

```
public DbEntityEntry Entry(object entity);

public DbEntityEntry<TEntity> Entry<TEntity>(TEntity entity) where
TEntity : class;
}
```

Como observará, por una lado los métodos **Entry** (en su versión no genérica y en su versión genérica) nos permitirán acceder o crear las referencias de nuestras entidades con respecto al OSM. Para ello se introduce un nuevo tipo a EF 4.1 conocido como *DbEntityEntry* cuya funcionalidad completa expondremos en profundidad en el siguiente punto.

Por su parte **ChangeTracker**, nos ofrece una vista mucho más simple del OSM ofreciéndonos acceso a los diferentes *DbEntityEntry* en un momento dado, sin necesidad de preocuparnos por estados y otros condicionamientos como tenemos que hacer si trabajáramos con el *ObjectStateManager* expuesto por *ObjectContext*.

Para terminar con *DbContext*, los distintos constructores nos permitirán establecer si este contexto trabajará con una base de datos específica, establecida en configuración o partir de un objeto *DbConnection*, tal y como ya hemos hecho en infinidad de ocasiones en todos los ejemplos anteriores.

4.2.- DbEntityEntry

Tal y como acabamos de comentar *DbContext* nos ofrece una visión mucho más simple de nuestro *ObjectStateManager*, aunque no por ello menos potente, por medio de los métodos *Entry* y de su propiedad *ChangeTracker*. Estos elementos, como comentamos, trabajan con un nuevo tipo introducido llamado **DbEntityEntry**, el cual, como veremos a continuación nos ofrecerá muchas más características que el ya conocido *ObjectStateEntry* de EF 4.1. Con el fin de presentar todas las características ofrecidas por *DbEntityEntry* y las distintas posibilidades que nos aporta, realizaremos una serie de ejemplos demostrativos basándonos en el siguiente modelo, simplificado para facilitar la comprensión, figura 21.

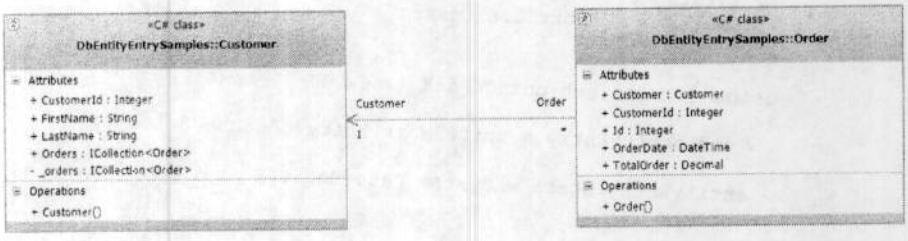


Figura 21.- Modelo simple para los ejemplos

4.2.1.- Gestión del estado

Lo primero que veremos es como *DbEntityEntry* nos permite una rápida y sencilla gestión del estado. El primer elemento diferenciador con respecto a *ObjectStateEntry* es que ahora no tendremos que buscar dentro de nuestro *ObjectStateManager* un elemento o adjuntarlo previamente para poder trabajar con él. Ahora solamente con el método *Entry* podemos hacer este trabajo, puesto que si se intenta obtener la referencia de un elemento no conocido por el contexto de trabajo, éste automáticamente la crea y nos la devuelve para nuestro posterior control.

A continuación podemos ver un ejemplo de esto mismo:

```
private static void DetachedEntityEntry()
{
    Customer detachedCustomer = new Customer()
    {
        CustomerId = 1,
        FirstName = "Unai",
        LastName = "Zorrilla Castro"
};
using (CRMContext unitOfWork = new CRMContext())
{
    var entityEntry = unitOfWork.Entry(detachedCustomer);
}
```

Si una vez que hemos recuperado el elemento *entityEntry* revisáramos el valor de su *EntityState* veríamos que este es **Detached**, es decir, que por ahora el contexto de trabajo no hace ningún seguimiento de él, incluso, aunque este elemento fuera un proxy y se pudieran determinar sus cambios automáticamente. Para que el contexto lo considerara como nuevo elemento, elemento para borrar o un elemento para modificar, solamente habría que establecer alguno de los distintos valores del enumerado *EntityState* a su propiedad **State**.

A continuación vemos una versión modificada del código anterior que nos permite establecer a nuestra entidad como una entidad modificada y por lo tanto que cuando la unidad de trabajo comprometa los cambios realice una actualización de la misma.

```
Customer detachedCustomer = new Customer()
{
    CustomerId = 1,
    FirstName = "Unai",
    LastName = "Zorrilla Castro"
};
using (CRMContext unitOfWork = new CRMContext())
{
    var entityEntry = unitOfWork.Entry(detachedCustomer);

    entityEntry.State = System.Data.EntityState.Modified;
```

```
        unitOfWork.SaveChanges();
    }
```

Por supuesto, con *DbEntityEntry* también podemos jugar con los valores originales y actuales, y por supuesto de una forma mucho más simple que como hacíamos hasta ahora. Las propiedades de solo lectura **CurrentValues** y **OriginalValues** nos ofrecen acceso a un elemento que almacena los distintos valores de las propiedades de las entidades, **DbPropertyValues**. Este tipo, nuevo también para EF 4.1, nos ofrece distintas posiblidades de acceso a los valores de las propiedades, así como a su modificación. A continuación mostramos una primera versión en la que establecemos los originales de una entidad:

```
using (CRMContext unitOfWork = new CRMContext())
{
   var entityEntry = unitOfWork.Entry(detachedCustomer);

   entityEntry.State = System.Data.EntityState.Modified;

entityEntry.OriginalValues["FirstName"] = "Unai";

   unitOfWork.SaveChanges();
}
```

Otra forma de establecer los originales, si no queremos hacer uso de los indexadores con los nombres de las propiedades entre comillas (algo que deberíamos evitar por otra parte), podemos utilizar dos técnicas distintas. La primera es mediante el método **SetValues** que nos ofrece la clase **DbPropertyValues**, método que nos permite establecer el objeto que representa los originales de una entidad.

```
using (CRMContext unitOfWork = new CRMContext())
{
   var entityEntry = unitOfWork.Entry(detachedCustomer);

   entityEntry.State = System.Data.EntityState.Modified;

entityEntry.OriginalValues.SetValues(new Customer() { CustomerId = 1,
FirstName = "Unai", LastName = "Gonzalez" });

    unitOfWork.SaveChanges();
}
```

La última técnica, y quizás la que más se utilice, consiste en acceder tanto a los valores originales como a los actuales como si fueran una entidad. Para ello la clase *DbPropertyValues* nos ofrece un método **ToObject** que nos permite transformar tanto los originales como los actuales en una instancia de un objeto que podremos tratar. En el siguiente fragmento puede observar un ejemplo de esto:

```
using (CRMContext unitOfWork = new CRMContext())
{
   var entityEntry = unitOfWork.Entry(detachedCustomer);

   entityEntry.State = System.Data.EntityState.Modified;
```

```
    Customer originals =
(Customer)entityEntry.OriginalValues.ToObject();

    originals.FirstName = "Unai";
    originals.LastName = "Gonzalez";


    unitOfWork.SaveChanges();
}
```

Como ya sabrá, la gestión de los valores actuales y originales es de vital importancia para resolver posibles problemas de concurrencia. A mayores del fácil acceso a estos elementos, algo que nos ha hecho ganar mucho ya con respecto a la anterior versión, la clase *DbEntityEntry* nos ofrece los métodos **GetDatabaseValues** y **Reload** que nos simplificarán en gran medida los procesos de resolución de concurrencia optimista. Para explicar estos métodos, haremos un pequeño cambio en el mapeo de nuestras entidades, estableciendo la propiedad *FirstName* de la entidad *Customer* como un *token* de concurrencia:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
                .Property(c => c.FirstName)
                .IsConcurrencyToken();

    base.OnModelCreating(modelBuilder);
}
```

Ahora intentaremos ver el método *Reload*, el cual tiene como función recargar los datos de la base de datos y establecerlos en los valores actuales de la entidad, es decir, sobrescribiendo los cambios que nosotros hubiéramos podido hacer. Fíjese que esta es la misma técnica que ya conocemos como *StoreWins*, vista en los puntos 2.5.1 y 2.5.4 del capítulo 4 del libro.

De forma general esta técnica se utiliza cuando se desea presentar al usuario un error y enseñarle los datos actuales de la entidad que ha incurrido en el fallo. El siguiente fragmento de código nos permite ver un pequeño ejemplo de esta casuística:

```
using (CRMContext unitOfWork = new CRMContext())
{
    Customer customer = unitOfWork.Customers.Find(1);

    customer.FirstName = "Lucia";

    /*simular concurrencia*/

                unitOfWork.Database.ExecuteSqlCommand("UPDATE
CUSTOMERS SET FirstName='Jose' WHERE CustomerId=1");

    /**********************/

    try
    {
        unitOfWork.SaveChanges();
```

```
        }
        catch (DbUpdateConcurrencyException ex)
        {
            ex.Entries.Single().Reload();

            //TODO:Devolver al usuario la entidad con los valores actuales
        }
    }
```

Nota: Fíjese como, para simular el problema de concurrencia, se ha hecho uso del método *ExecuteSqlCommand*. Método cuyas acciones (modificaciones, borrados etc.) no son tenidos en cuenta por el *change tracker* de nuestro contexto de trabajo.

Por supuesto, la opción de *StoreWins* no es siempre la que queremos implementar. En EF 4.1 ya conocemos la existencia de otra opción como *ClientWins*. Para esta opción nuestro propósito no es más que establecer los valores originales de una entidad (son los que se utilizan para los distintos fragmentos del segmento SET) con los que hay en un momento dado en la base de datos, y así evitar que surjan los problemas de concurrencia haciendo que nuestros valores actuales ganen (*Last wins*). *DbEntityEntry* nos ofrece, como ya comentamos, el método **GetDatabaseValues**, que como ya se imaginará nos permite obtener los valores almacenados en la base de datos para una entidad. Ahora, sumado esto a que podemos establecer éstos como los originales de una forma sencilla, ya tenemos una forma trivial de implementar nuestro **ClientWins**.

A continuación, podemos ver el mismo ejemplo utilizando esta nueva técnica:

```
using (CRMContext unitOfWork = new CRMContext())
{
    Customer customer = unitOfWork.Customers.Find(1);
    customer.FirstName = "Lucia";

    /*simular concurrencia*/

     unitOfWork.Database
               .ExecuteSqlCommand("UPDATE CUSTOMERS SET
FirstName='Jose' WHERE CustomerId=1");

    /**********************/

    try
    {
        unitOfWork.SaveChanges();
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var entry = ex.Entries.Single();
        var dbValues = entry.GetDatabaseValues();

        entry.OriginalValues.SetValues(dbValues);

        unitOfWork.SaveChanges();
```

```
        }
    }
```

4.2.2.- Gestión de las referencias

DbEntityEntry, además de las distintas facilidades que nos proporciona para la gestión del estado también nos permite manejar la carga y el trabajo con las asociaciones de unas entidades con otras. Para ello nos ofrece los métodos **Collection<T>** y **Reference<T>**, los cuales nos permiten acceder a las asociaciones de cardinalidad múltiple y simple respectivamente.

En el siguiente fragmento de código puede observarse como hacer uso del método *Collection<T>* para cargar la navegación de nuestra entidad *Customer*:

```
Customer detachedCustomer = new Customer()
{
    CustomerId = 1,
    FirstName = "Unai",
    LastName = "Zorrilla Castro"
};

using (CRMContext unitOfWork = new CRMContext())
{
    var entityEntry = unitOfWork.Entry(detachedCustomer);

    entityEntry.State = System.Data.EntityState.Modified;

    var collection = entityEntry.Collection<Order>(c => c.Orders);

    if ( !collection.IsLoaded)
        collection.Load();
}
```

Nota: Fíjese en le código anterior como el método *Collection<T>* nos permité pasar como parámetro una expresión lambda que represente a la colección que deseeamos cargar en vez de tener que escribir una cadena de caracteres con el nombre de la misma. Para el caso del método *Reference* el trabajo es idéntico.

Uno de los puntos más interesantes con respecto a los métodos *Collection<T>* y *Reference<T>* es que los tipos devueltos nos ofrecen acceso al elemento **IQueryable<T>** por medio de un método llamado **Query**. Gracias a esto, las colecciones que carguemos podrían, por ejemplo, restringirse a una consulta específica, algo que no podemos hacer con **Include** y que solemos hacer con el truco de devolver un tipo anónimo con las distintas partes que deseamos carga. A continuación podemos ver un sencillo ejemplo de esto:

```
using (CRMContext unitOfWork = new CRMContext())
```

```
{
    var entityEntry = unitOfWork.Entry(detachedCustomer);

    entityEntry.State = System.Data.EntityState.Modified;

    var collection = entityEntry.Collection<Order>(c => c.Orders);

    if (!collection.IsLoaded)
    {
       collection.Query()
                 .Where(o=>o.TotalOrder > 1000)
                 .Load();
    }
}
```

Nota: Gracias al método *Query* se nos abren otras puertas, como por ejemplo la de poder contar el numero de elementos de una navegación sin tener que acceder directamente a su *DbSet* por medio del contexto. Un ejemplo de esto sería:

```
int count = collection.Query().Count();
```

4.3.- DbSet - IDbSet

Al igual que DbContext supone un adaptador sobre nuestro *ObjectContext*, *DbSet* es una adaptación de nuestros *ObjectQuery* de Entity Framework, dándonos una superficie de trabajo más simple, así como alguna funcionalidad extra que veremos en las siguientes líneas. Esta nueva superficie de trabajo proporcionada por *DbSet* viene dada por su contrato base **IDbSet<T>,** cuya firma principal podemos ver a continuación en el siguiente fragmento de código:

```
public interface IDbSet<TEntity> : IQueryable<TEntity>,
IEnumerable<TEntity>, IQueryable, IEnumerable
   where TEntity : class
{
 ObservableCollection<TEntity> Local { get; }
 TEntity Add(TEntity entity);
 TEntity Attach(TEntity entity);
 TEntity Create();
 TDerivedEntity Create<TDerivedEntity>()
     where TDerivedEntity : class, TEntity;
 TEntity Find(params object[] keyValues);
   TEntity Remove(TEntity entity);
}
```

Fíjese como esta firma es muy similar a la que nos ofrece *IObjectSet<TEntity>*, vista anteriormente en este mismo libro. Ahora *IDbSet* en vez de *AddObject* y *DeleteObject* nos ofrece métodos con un nombre más cercanos a un repositorio, **Add** y

Remove. Lógicamente, a mayores de este en principio "insignificante cambio de nombre" también hay unas cuantas nuevas funcionalidades expuestas en este contrato.

La primera de dichas funcionalidades está dada por los métodos **Create** y **Create<TDereviedEntity>**, gracias a los cuales, este conjunto nos permitirá crear entidades "tipo proxy" de los elementos que el *IDbSet<TEntity>* maneje. El siguiente fragmento nos permite comprobar como con el método *Create* podemos crear la versión proxy de un tipo dado:

```
using (CRMContext unitOfWork = new CRMContext())
{
   var customer = unitOfWork.Customers.Create();
}
```

Como habrá imaginado la otra sobrecarga del método *Create* nos permite realizar esta misma tarea para alguno de los tipos de una jerarquía, es decir, crear un proxy para alguno de los tipos que hereden de una entidad base.

La segunda de las funcionalidades que podemos ver de una pasada en nuestro contrato *IDbSet* es la referente al método de búsqueda **Find**. Gracias a éste podremos hacer la búsqueda de una entidad en función de los valores de su clave primaria. Si recuerda del punto 3.5.1 de este mismo capítulo, en la sección de descubrimiento de claves, vimos como EF 4.1 obligaba a usar el atributo *Column* y la especificación de *Order* cuando se mapeaban claves compuestas (esto no era necesario si se hacía por código), este hecho, como comentamos se debe que este orden será el orden tomado por este método Find para realizar las búsquedas de entidades por clave.

```
using (CRMContext unitOfWork = new CRMContext())
{
   var customer = unitOfWork.Customers.Find(1);
}
```

Para terminar con las nuevas funcionalidades expuestas con *IDbSet<T>* hablaremos de los datos locales a un conjunto, funcionalidad demandada a menudo en los foros de Entity Framework pero que hasta ahora no había tenido respuesta. En algunos escenarios de trabajo, sobre todo para aquellos que utilizan las unidades de trabajo con un ciclo de vida largo, puede resultar ventajoso poder hacer consultas a los datos locales de un *DbSet*, es decir, poder hacer consultas sobre aquellos elementos que se han traido de la base de datos y se están *trackeando* actualmente. Hasta ahora para poder realizar esta funcionalidad teníamos que crearnos nuestras propias piezas de código que interrogaran al *ObjectStateManager* de EF y nos dieran esta funcionalidad.

Ahora, con nuestro nuevo tipo *DbSet<TEntity>* ya tenemos este trabajo hecho, puesto que el mismo nos proporciona una propiedad llamada **Local**, la cual nos devuelve en forma de una **ObservableCollection<TEntity>** las entidades que se están siguiendo actualmente, simpre y cuando su estado no sea "Deleted".

```
public interface IDbSet<TEntity> : IQueryable<TEntity>,
IEnumerable<TEntity>, IQueryable, IEnumerable where TEntity : class
{
 // Omitido por brevedad
```

```
    ObservableCollection<TEntity> Local { get; }
}
```

Para mostrar las funcionalidades de *Local*, supondremos que partimos de un modelo con una serie de entidades almacenadas en su base de datos. Sobre este modelo, realizaremos una pequeña consulta y después observaremos los datos locales del *DbSet<TEntity>* que se haya consultado.

Nota: Si no quiere partir de una base de datos con una pregeneración de datos puede crear un inicializador nuevo, por ejemplo basándose en la clase base *DropCreateDatabaseAlways<TContext>* y sobreescribir el método *Seed* para incluir los datos de prueba que considere. En el ejemplo sobre *Local* que acompaña a este capítulo se ha realizado esta tarea.

```
using (CRMContext context = new CRMContext())
{
    var db = context.Customers
                    .ToList();

    Dump("Local", context.Customers.Local);
    Dump("DB    ", db);
}
```

El método **Dump** en este código únicamente muestra los *Id* de los elementos de cada una de las colecciones, Local y DB. Para este primer ejemplo la salida es idéntica en ambos y sería algo como sigue:

```
Local
    Id:1
    Id:2
DB
    Id:1
    Id:2
```

La salida anterior es lógica teniendo en cuenta que *Local* nos permite acceder a los datos consultados y que se están "trackeando" por el contexto de trabajo. Para comprobar que este comportamiento es correcto, probaremos a realizar la consulta indicando al contexto que no debe de "trackear o seguir" los resultados obtenidos, es decir, hacer lo mismo que podíamos hacer con nuestros *IObjectSet* y su propiedad *MergeOption*:

```
private static void ShowLocalData()
{
    using (CRMContext context = new CRMContext())
    {
        var db = context.Customers
                        .AsNoTracking()
                        .ToList();
```

```
        Dump("Local", context.Customers.Local);
        Dump("DB    ", db);
    }
}
```

Ahora con este cambio, **AsNoTracking**(), el resultado sería diferente puesto que como puede ver a continuación en el resultado, los datos locales ya no continen ningún elemento.

```
Local

DB
    Id:1
    Id:2
```

A mayores de la condición impuesta de que el contexto esté realizando un seguimiento de las entidades para que éstas se incluyan como datos locales a un *DbSet*, también se impone la condición de que las mismas no se hayan marcado para eliminarse: su estado sea distinto de *Deleted*.

Si modificamos el código anterior para que por ejemplo se marque para borrar a la entidad situada en posición de índice 0, veríamos como su referencia ya no aparecería dentro de los datos locales de la entidad.

```
private static void ShowLocalDataWithDeleted()
{
    using (CRMContext context = new CRMContext())
    {
        var db = context.Customers
                        .ToList();

        context.Customers.Remove(db[0]);

        Dump("Local", context.Customers.Local);
        Dump("DB    ", db);
    }
}
```

```
Local
    Id:2
DB
    Id:1
    Id:2
```

Para los datos agregados a un *DbSet* la situación es al revés que esta anterior. En cuanto se agrega un elemento a un *DbSet* (sin comprometerlo en la base de datos), la referencia a este elemento aparece dentro de los datos locales pero no en los de la consulta, lógicamente por otra parte. A continuación se puede ver un fragmento de código y el resultado que ilustra esta casuística.

```
using (CRMContext context = new CRMContext())
{
    Customer customer = new Customer()
    {
```

```
            FirstName = "Lucia",
            LastName = "Zorrilla"
        };

        context.Customers.Add(customer);


        var local = context.Customers.Local;
        var db = context.Customers.ToList();

        Dump("Local", local);
        Dump("DB    ", db);
    }
```

```
Local
    Id:0
    Id:1
    Id:2
DB
    Id:1
    Id:2
```

Tal y como comentamos al principio del párrafo sobre datos locales, éstos se suelen utilizar a menudo en escenarios de contextos de vida corto, dónde presumiblemente estemos enlazando a datos directamente nuestras entidades del modelo. Para estos casos hacer databinding a los datos en *Local* (por ejemplo desde WPF), nos ahorra cierto trabajo puesto que como ya habrá visto y observado *Local* es una *ObservableCollection* y por lo tanto candidata perfecta para esta tarea. Esta colección devuelta, está además en perfecta sincronía con los datos consultados, por ello no tendremos que hacer ningún trabajo extra en cuanto al refresco de sus datos.

Para el caso de Windows Forms, dónde *ObservableCollection* no tiene la misma implicación que en WPF, tenemos disponible un método extensor llamado **ToBindingList** que nos permitirá transformar el resultado de esta propiedad a algo más amigable para el enlace datos en esta tecnología.

```
public static BindingList<T> ToBindingList<T>(this
ObservableCollection<T> source) where T : class;
```

4.4.- Validación de entidades

Con el aditivo de EF 4.1 es la primera vez que disponemos en ADO.NET Entity Framework de algo relativo a la validación de entidades. Este tema no tiene porqué ser tan nimio como pueda parecer en un principio, puesto que -aunque no lo parezca- puede llegar a tener hasta un coste monetario relativamente importante. Desde ahora en adelante trataremos de diferenciar las validaciones del modelo, en cuanto a la estructura de una base de datos (es decir, valores nulos o no nulos, etc.) con la validación de conceptos del negocio (como por ejemplo podría ser que el valor de una propiedad tuviera que cumplir cierta expresión regular). En general la validación de un modelo (intentar incluir una propiedad nula en un campo requerido, por ejemplo) suele causar

muchos *roundtrips* innecesarios. Por supuesto, además de la latencia que esto podría suponer para los usuarios del sofware en cuestión, en entornos como Azure dónde pagamos por el uso de la base de datos esto podría tener consecuencias económicas para el producto, como ya se ha comentado.

Bien, empecemos por el principio y veamos como se implementan las validaciones y las consideraciones que tenemos que tener en cuenta, tanto para las validaciones de modelo como para las personalizadas de nuestro negocio que querramos implementar.

Para ello, partiremos del modelo más simple que podamos, que contenga únicamente la siguiente entidad. Fíjese que se ha forzado para incluir alguna propiedad que nos pueda dar algún juego extra en la validación:

```
public class Customer
{
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string DNI { get; set; }
    public string Mail { get; set; }
    public string ZipCode { get; set; }
}
```

Ejemplos típicos de validaciones del modelo para este caso podrían ser elementos como tamaño máximo de longitudes de cadena, valores *nullables* o no (recuerde que el tipo *string* es un tipo por referencia), etc. Para este trabajo desde el equipo de ADO.NET EF y puesto que se decidió compatibilizar el uso de Data Annotations con el de modelado, nos permite establecer estos criterios de validación por los métodos de mapeo que conocemos como **IsRequired**, **HasMaxLenght**, etc, o bien usando las anotaciones como ya sabemos de un punto anterior en este mismo capítulo.

A continuación, en las siguientes líneas, puede ver un ejemplo de mapeo en el que se establecen como requeridas y con un máximo de longitud las propiedades *FirstName* y *LastName* de nuestra entidad *Customer*.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
                .Property(p => p.FirstName)
                .IsRequired()
                .HasMaxLength(10);

    modelBuilder.Entity<Customer>()
                .Property(p => p.LastName)
                .IsRequired()
                .HasMaxLength(20);
}
```

Si ahora intentáramos guardar una entidad que incumpliera alguno de estos criterios, por ejemplo, la siguiente instancia:

```
using (CRMContext context = new CRMContext())
{
    Customer customer = new Customer()
```

```
        {
            Mail ="unai@plainconcepts.com"
        };

        context.Customers.Add(customer);
        context.SaveChanges();
    }
```

Obtendríamos una excepción lanzada en el método *SaveChanges* de nuestro contexto, de tipo **DbEntityValidationException**. Esta excepción, a mayores de lo que le impone su clase base, nos ofrece una propiedad **EntityValidationErrors** con la colección de errores de validación de las entidades que han entrado en juego en el proceso de compromiso de los cambios.

```
public IEnumerable<DbEntityValidationResult> EntityValidationErrors {
get; }
```

Estos errores de validación vienen dados por el tipo **DbEntityValidationResult**, tipo que a mayores de otorgarnos la referencia a la entidad con el problema de validación por medio de su propiedad **Entry**, nos ofrece una lista de mensajes de error obtenidos de los atributos de validación en sus mensajes por defecto o por medio de mensajes personalizados que veremos cómo incluir más adelante.

```
public class DbEntityValidationResult
{
public DbEntityEntry Entry { get; }

public bool IsValid { get; }

public ICollection<DbValidationError> ValidationErrors { get; }

}
```

En nuestro ejemplo, si quisiésemos atrapar nuestros mensajes de error, solamente tendríamos que manejar la excepción anterior y aplanar los resultados como necesitásemos. A continuación puede ver un ejemplo sencillo de este trabajo dónde se realiza una salida por pantalla del estado de las entidades que incurren en un error de validación así como de las propiedades afectadas junto con su mensaje de error.

```
try
{
    context.SaveChanges();
}
catch (DbEntityValidationException ex)
{
    Dump(ex);
}
```

```
private static void Dump(DbEntityValidationException ex)
{
   foreach (var item in ex.EntityValidationErrors)
   {
      Console.WriteLine("Errores de validación para una entidad con
```

```
estado:{0}",item.Entry.State);

        Console.WriteLine("Los errores de validación son:");

        foreach (var ve in item.ValidationErrors)
        {
            Console.WriteLine("\tPropiedad:{0}
Error:{1}",ve.PropertyName,ve.ErrorMessage);
        }
    }
}
```

Como habrá visto, no ha hecho falta ningún trabajo extra para habilitar la validación de nuestras entidades, y automáticamente esta validación se ejecuta en el proceso de guardado. Si no quisiésemos tener habilitada esta validación, activada por defecto como acabamos de decir, solamente tendríamos que ir a la propiedad de configuración de nuestra unidad de trabajo y deshabilitarla.

```
context.Configuration.ValidateOnSaveEnabled = false;
```

Igualmente si no deseamos esperar hasta el guardado para realizar las validaciones de las entidades podemos hacer éstas "on-demand" de dos formas distintas. La primera es llamando al método del contexto de trabajo **GetValidationResult**, el cuál, nos ofrecerá la misma información que la excepción sin esperar a que esta tenga que producirse:

```
var validationErrors = context.GetValidationErrors();
```

La segunda forma nos permitirá realizar la validación entidad por entidad de una forma simple, para ello, solamente tendremos que acceder al *DbEntityEntry* de la entidad y llamar al método *GetValidationResult*.

```
var vresult = context.Entry(customer)
                     .GetValidationResult();
```

Tal y como comentamos, la validación de estos elementos del modelo también puede realizarse mediante atributos (estos elementos de hecho los hemos visto durante el punto referido al *mapping* de las entidades). El mismo ejemplo que el anterior lo podríamos haber hecho como sigue:

```
public class Customer
{
    public int CustomerId { get; set; }

    [Required()]
    [MaxLength(10)]
    public string FirstName { get; set; }

    [Required()]
    [MaxLength(10)]
    public string LastName { get; set; }
```

```
        public string DNI { get; set; }
        public string Mail { get; set; }
        public string ZipCode { get; set; }
    }
```

Permítame recordarle en este punto algo sobre lo que hablamos anteriormente en referencia al uso de anotaciones dependientes de **EntityFramework.dll,** como es el caso de *MaxLength* dentro del modelo de entidades y, con esto, la posible rotura del principio ignorancia de la persistencia. En este caso, quizás hubiera sido mejor usar *StringLength* en lugar de *MaxLength.*

Un punto a favor del uso de anotaciones (siempre procurando que no sean dependientes de ADO.NET EF) es que nos ofrecen la posibilidad de establecer los mensajes de error como mensajes en recursos incrustados, haciendo nuestras aplicaciones localizables en cuanto a las validaciones. En el siguiente fragmento de código puede ver el uso de la anotación de validación *Required* haciendo uso de un fichero de recursos llamado *ValidationMessages* y una clave de este fichero de recursos llamada *validation_FirstNameIsRequired*:

```
[MaxLength(10)]
[Required(
ErrorMessageResourceName="validation_FirstNameIsRequired",
ErrorMessageResourceType=typeof(ValidationMessages))]
public string FirstName { get; set; }
```

De las posibles anotaciones que podemos usar para cubrir la validación de nuestro modelo, sin ser dependientes de EF u otras tecnologías como MVC, tenemos, como ya hemos mencionado, todas aquellas que heredan de *ValidationAttribute*. Si mirarmos la jerarquía con nuestra herramienta favorita podríamos ver como, por defecto, disponemos de las siguientes:

Tabla 10.- Anotaciones de validación de System.ComponentModel.DataAnnotations

Validación	Propósito
StringLengthAttribute	Validación del tamaño máximo del número de caracteres de un string
RequiredAttribute	Validación de un elemento como requerido
RegularExpressionAttribute	Validación que asegura que el valor cumple una determinada expresión regular.
RangeAttribute	Validación que nos permite asegurar que el valor de una propiedad está en un rango determinado.
CustomValidationAttribute	Anotación que nos permite establecer una validación personalizada

La anotación *CustomValidationAttribute* es especial puesto que nos permite delegar en un validador externo. Con esta anotación podremos especificar la clase y el método (nos obliga lógicamente a que sea una clase pública y un método publico y estático)

que queremos usar para realizar la validación. Con el fin de poner un ejemplo intentaremos crear un validador para el DNI español y posteriormente asignárselo a la entidad.

El código del validador sería algo como lo siguiente:

```
public static ValidationResult ValidateDNI(string dni)
{
    //TODO: Validate DNI
  bool result;

  if (!result)
  {
     return new ValidationResult("DNI incorrecto", new string[] {
"DNI" });
  }
  else
     return null;
}
```

Y, por su parte, la decoración de nuestra entidad ahora contendría lo siguiente en su propiedad DNI;

```
[CustomValidation(typeof(CustomerValidation),"ValidateDNI")]
public string DNI { get; set; }
```

Con estos sencillos pasos, cuando se realiza la validación de una entidad, en el método *SaveChanges* u *On-Demand*, EF se encargará de llamar a este validador y ofrecer los resultados de igual forma que para el resto de anotaciones.

Nota: Aunque no se ha comentado, comviene destacar aquí que ADO.NET EF 4.1 también soporta las anotaciones incluidas en los tipos complejos, de tal forma que el sistema es capaz de hacer un traverse entre los tipos y así validar también las propiedades incluidas en ellos, si hay anotaciones lógicamente.

Realmente, con lo que hemos hecho hasta ahora, ya hemos ganado mucho. Hemos incorporado a nuestras entidades un sistema de validación del modelo. Además el sistema de validación que hemos usado es independiente de la tecnología, o podemos hacerlo si no usamos las anotaciones de EF. De hecho, estas mismas anotaciones de validación que hemos incorporado a las entidades son perfectamente compatibles con las validaciones que usa y entiende ASP.NET MVC sin necesidad de hacer absolutamente nada.

Aún así, el trabajo no queda solamente aquí, puesto que a mayores de las validaciones de las propiedades puede que necesitemos realizar validaciones lógicas, de acuerdo a alguna regla de negocio que pueda involucrar a una o a más propiedades. Generalmente a este tipo de validaciones se les suele llamar "validaciones a nivel de clase". Para esta tarea el equipo de trabajo de ADO.NET Entity Framework ha dado soporte a las validaciones por medio de la interfaz **IValidatableObject**, permitiendo

que nuestras entidades puedan implementar esta interfaz para realizar validaciones más complejas de la entidad.

Una nota importante sobre estas validaciones es que las mismas solamente se procesan cuando no hay ningún error a nivel de anotación del modelo. Es decir, solamente se tienen en cuenta si el resto de anotaciones se han validado con un resultado positivo.

En el siguiente fragmento de código se puede observar una implementación posible -algo estúpida por otra parte :-)- para esta interfaz en nuestra entidad *Customer*:

```
public IEnumerable<ValidationResult> Validate(ValidationContext
validationContext)
{
   if (Mail == "unai@plainconcepts.com"
      &&
      ZipCode == "1111")
   {
      yield return new ValidationResult("This is a stupid
validation!", null);
   }
}
```

5.- CONCLUSIONES

A lo largo de los cuatro puntos anteriores hemos visto los principales elementos incorporados con el aditivo de EF 4.1, intentando desgranar las nuevas posibilidades que estos nos dan y las mejores prácticas para poner estos elementos en un entorno real. La verdad, estimado lector, es que muchos esperamos con ansia estas nuevas funcionalidades, así como algunas de las que aún no disponemos pero de las que vemos sus primeros pasos.

APÉNDICE

Fundamentos de LINQ

Nota: Aunque en este capítulo vea referencias a C# 3.0, no se trata de un error aunque vayamos en una versión posterior del lenguaje. Se indica esta versión puesto que en realidad LINQ tiene su origen en ella, y no en la actual versión del lenguaje.

1.- INTRODUCCIÓN

Las consultas integradas en el lenguaje (la traducción literal de Language Integrated Query, origen del acrónimo LINQ) constituyen uno de los avances más significativos de los últimos tiempos en el área de los lenguajes y sistemas de programación. Están jugando un papel fundamental en la tarea de reducir a su mínima expresión los efectos del fenómeno conocido como "**desajuste de impedancias**" (*impedance mismatch*), que nos imponía la necesidad de utilizar al desarrollar aplicaciones no sólo nuestro lenguaje de programación habitual, sino además toda una serie de otros lenguajes diferentes para acceder a las más diversas fuentes de datos, como SQL o XPath/XQuery, cuyas sentencias se "incrustan" actualmente de manera literal dentro del código C# o Visual Basic.

Gracias a LINQ, parte integral de las nuevas versiones de estos lenguajes a partir de .NET Framework 3.5, podremos realmente empezar a crear aplicaciones que contengan única y exclusivamente código .NET, utilizando una sintaxis clara y natural para acceder a esos almacenes de datos, y dejando al compilador y las librerías de soporte la tarea de generar esas construcciones "foráneas".

1.1.- Presentación de LINQ

LINQ es una combinación de extensiones al lenguaje y librerías de código manejado que permite expresar de manera uniforme las consultas sobre colecciones de datos provenientes de las más disímiles fuentes: objetos en memoria, documentos XML o bases de datos relacionales, entre otros.

El propio significado del acrónimo (Consultas Integradas en el Lenguaje) ofrece la clave para comprender el objetivo principal de LINQ: permitir que el programador exprese las consultas sobre datos de diversa procedencia utilizando recursos de su propio lenguaje de programación. Las ventajas fundamentales de esta posibilidad son las siguientes:

- A la hora de expresar sus consultas, el programador podrá aprovechar todas las facilidades que el compilador y el entorno integrado ofrecen (verificación de sintaxis y de tipos por parte del compilador, ayuda IntelliSense dentro del entorno, acceso a metadatos por parte de ambos).
- LINQ permitirá, si no eliminar completamente, sí reducir en gran medida el ya mencionado fenómeno del "desajuste de impedancias" que provocan las diferencias entre los modelos de programación que proponen los lenguajes de propósito general y los lenguajes de consulta sobre bases de datos relacionales (SQL), documentos XML (XPath/XQuery) y otros almacenes de datos.
- Por último, otra ventaja importante de LINQ es que permitirá elevar el nivel de abstracción y claridad de la programación de las consultas. Por ejemplo, un programador que necesite hoy en día acceder a una base de datos, deberá especificar en un plan meticuloso **cómo** recuperar los datos que necesita; las expresiones de consulta, en cambio, son una herramienta mucho más declarativa, que permite en gran medida indicar únicamente **qué** se desea obtener, dejando al motor de evaluación de expresiones los detalles sobre cómo lograr ese objetivo.

El otro elemento clave de la tecnología LINQ es su arquitectura abierta, que hace posible la extensibilidad. La semántica de los operadores en las expresiones de consulta no está en modo alguno "cableada" en el lenguaje, sino que puede ser modificada (o extendida) por librerías de Microsoft o de terceros fabricantes para el acceso a fuentes de datos específicas. El propio Microsoft pone a nuestra disposición, además del mecanismo estándar para aplicar LINQ a *arrays* y colecciones genéricas (**LINQ to Objects**), al menos otras cuatro tecnologías: por una parte, **LINQ to XML**, para ejecutar consultas integradas sobre documentos XML y **LINQ to DataSets**, para la ejecución de consultas al estilo LINQ contra conjuntos de datos tipados y no tipados; por la otra, **LINQ to SQL** y **LINQ to Entities**, para hacer posible las consultas integradas contra bases de datos relacionales.

Pero la arquitectura de las herramientas que se ponen a nuestra disposición con LINQ es tal, que ha hecho posible la aparición de tecnologías

("proveedores") que simplifican y uniformizan el acceso a otras fuentes de datos: **LINQ to Amazon** y **LINQ to SharePoint** son dos de los ejemplos más contundentes que se pueden encontrar al navegar por la red.

La arquitectura de LINQ puede ser descrita gráficamente de la siguiente forma:

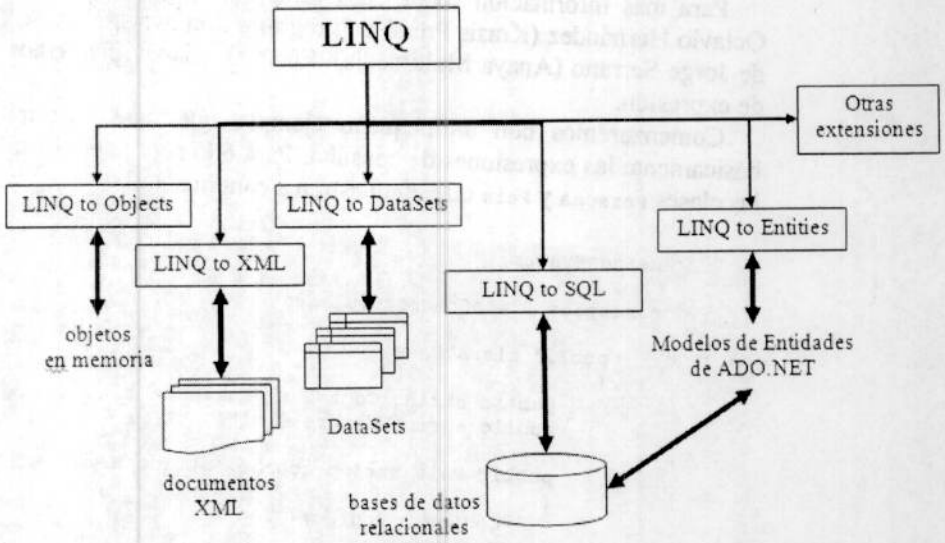


Figura A.1.- Representación gráfica de la arquitectura de LINQ

1.2.- Las expresiones de consulta

Las expresiones de consulta (*query expressions*) son el mecanismo mediante el cual cobra vida la tecnología LINQ. No son otra cosa que expresiones que responden a una nueva sintaxis que se ha añadido a C# 3.0 y Visual Basic 9.0 y que pueden actuar sobre cualquier objeto que implemente la interfaz genérica `IEnumerable<T>` (en particular, los *arrays* y colecciones de .NET 2.0 y superiores implementan esta interfaz), transformándolo mediante un conjunto de operadores en otros objetos, generalmente (pero no siempre) objetos que implementan esa misma interfaz.

Nota: Al referirnos a un objeto que implementa la interfaz *IEnumerable<TEntity>* utilizaremos frecuentemente el término "secuencia"; no sería correcto decir *array* ni colección, y "enumerable" (que en el fondo es lo que son, objetos que se pueden enumerar) podría causar confusión con los tipos enumerados **enum**.

Las expresiones de consulta se apoyan a su vez en otras novedades incorporadas a C# 3.0 y Visual Basic 9.0, como son:

- Declaración implícita del tipo de variables locales
- Inicializadores de objetos y colecciones
- Tipos anónimos
- Métodos extensores
- Expresiones lambda
- Árboles de expresiones

Para más información sobre estas características, consulte "C# 3.0 y LINQ", de Octavio Hernández (Krasis Press) o "Programación avanzada con Visual Basic 2008", de Jorge Serrano (Anaya Multimedia). En lo sicesivo utilizaremos C# como vehículo de expresión.

Comenzaremos con un pequeño ejemplo que nos mostrará en qué consisten básicamente las expresiones de consulta. Para ello supondremos que tenemos definida las clases `Persona` y `Pais` que se muestran a continuación.

```
using System;

namespace PlainConcepts.Clases
{
    public class Pais
    {
        public string Codigo { get; set; }
        public string Nombre { get; set; }

        public Pais(string codigo, string nombre)
        {
            Codigo = codigo;
            Nombre = nombre;
        }
    }

    public enum SexoPersona { Mujer, Varón }

    public class Persona
    {
        #region Propiedades
        public string Nombre { get; set; }
        public string CodigoPaisNac { get; set; }
        public DateTime? FechaNac { get; set; }
        public SexoPersona? Sexo { get; set; }

        public int? Edad
        {
            get
            {
                if (FechaNac == null)
                    return null;
                int dias = (int)
                    ((DateTime.Today - FechaNac.Value).TotalDays);
                return dias / 365;
            }
```

```
        }
        #endregion

        #region Constructores
        public Persona() { }
        public Persona(string nombre, string pais): this()
        {
            this.Nombre = nombre;
            this.CodigoPaisNac = pais;
        }
        #endregion

        #region Métodos
        public override string ToString()
        {
            return
                (Nombre == null ? "???" : Nombre) +
                (CodigoPaisNac == null ? " (??)" :
                    " (" + CodigoPaisNac + ")") +
                (Sexo == null ? "" :
                    (Sexo.Value == SexoPersona.Mujer ? " (M)" : "
(V)")) +
                    (FechaNac == null ? "" :
                        " (" + FechaNac.Value.ToString("dd/MM/yyyy")
+
                        ")");
        }
        #endregion
    }
}
```

Adicionalmente, la clase `Datos` que se presenta a continuación define propiedades estáticas que nos suministran algunos datos de prueba:

```
using System;
using System.Collections.Generic;

namespace PlainConcepts.Clases
{
    public static class Datos
    {
        public static List<Pais> Paises = new List<Pais> {
            new Pais("ES", "ESPAÑA"),
            new Pais("CU", "CUBA"),
            new Pais("RU", "RUSIA"),
            new Pais("US", "ESTADOS UNIDOS")
        };

        public static List<Persona> Personas = new List<Persona> {
            new Persona {
                Nombre = "Diana",
                CodigoPaisNac = "ES",
                FechaNac = new DateTime(1996, 2, 4),
                Sexo = SexoPersona.Mujer
            },
            new Persona {
                Nombre = "Dennis",
                CodigoPaisNac = "RU",
                FechaNac = new DateTime(1983, 12, 27),
```

```
                Sexo = SexoPersona.Varón,
            },
            new Persona {
                Nombre = "Claudia",
                CodigoPaisNac = "CU",
                FechaNac = new DateTime(1989, 7, 26),
                Sexo = SexoPersona.Mujer,
            },
            new Persona {
                Nombre = "Jennifer",
                CodigoPaisNac = "CU",
                FechaNac = new DateTime(1982, 8, 12),
                Sexo = SexoPersona.Mujer,
            }
        };
    }
}
```

Suponga que, a partir de los datos anteriores, deseamos obtener una colección con los nombres y edades de los mayores de 20 años que pertenecen a la lista original, con los nombres convertidos a mayúsculas. Los objetos de la colección resultante deberán, además, aparecer en orden alfabético.

Si no conoce todavía las maravillas de las consultas integradas, casi con total certeza lo primero que le vendrá a la mente será buscar un mecanismo para recorrer la colección original, creando nuevos objetos que contengan las características necesarias de las personas que cumplen con los requisitos solicitados; para luego ordenar alfabéticamente por los nombres esa colección en una segunda "pasada". Gracias a LINQ, ¡esa vía se convierte en anacrónica!

En C# 3.0 existe una manera mucho más clara y elegante de obtener ese resultado de un solo golpe:

```
using System;
using System.Linq;

namespace Demo1
{
    using PlainConcepts.Clases;

    class Program
    {
        static void Main(string[] args)
        {
            var mayores20 = from h in Datos.Personas
                            where h.Edad > 20
                            orderby h.Nombre
                            select new
                            {
                                Nombre = h.Nombre.ToUpper(),
                                Edad = h.Edad
                            };

            foreach (var h in mayores20)
                Console.WriteLine(h.Nombre + " (" + h.Edad + ")");
        }
    }
}
```

La semántica de la sentencia de asignación que contiene la consulta integrada será intuitivamente clara para cualquiera que esté familiarizado con la sentencia SELECT de SQL. Lo menos habitual es el hecho de que la cláusula `select`, en la que se especifica qué se desea obtener, va situada al final, a diferencia de SQL. La razón para ese cambio de posición tiene bastante lógica: si `select` estuviera al principio, sería imposible ofrecer al programador ayuda *Intellisense* a la hora de teclear los datos a seleccionar, porque aún no se habría indicado la colección de datos sobre la que se va a ejecutar la consulta. Teniendo a ese "operando" en primer lugar, es fácil para el entorno integrado ofrecer ayuda a lo largo de todo el proceso de tecleo de la expresión: si `Datos.Personas` es de tipo `List<Persona>` (y por lo tanto, `IEnumerable<Persona>`), entonces `h` (el nombre elegido por nosotros para la variable que irá refiriéndose sucesivamente a cada uno de los elementos de la secuencia) es de tipo `Persona`, y el sistema podrá determinar si es correcta o no cualquiera de las expresiones que aparecen en las cláusulas `where`, `orderby`, etc.

En la sentencia se hace uso de tres características aparecidas con C# 3.0:

- **Tipos anónimos**: como la estructura del conjunto de resultados deseado no coincide con la del tipo original de los elementos, en la cláusula `select` de la expresión se define *ad hoc* un nuevo tipo de datos que será generado automáticamente por el compilador. El resultado que produce la expresión de consulta es una colección de objetos de este tipo anónimo.
- Ya de por sí los tipos anónimos se apoyan en **inicializadores de objetos**, que permiten asignar los valores iniciales a los campos de los objetos del tipo anónimo.
- Por último, el objeto resultante de la ejecución de la consulta se asigna a una **variable cuyo tipo se infiere automáticamente**. Si bien en algunos casos el uso de la determinación automática del tipo de una variable local es solo una comodidad, en combinación con los tipos anónimos su uso es simplemente imprescindible. También se utiliza esta característica posteriormente al declarar la variable utilizada para recorrer la colección resultante mediante el bucle `foreach`.

A continuación veremos cómo por detrás del telón están entrando en funcionamiento también otras dos de las novedades de C# 3.0: las **expresiones lambda** y los **métodos extensores.**

1.3.- Reescritura de expresiones de consulta

¿Qué hace el compilador cuando encuentra una expresión de consulta? Las reglas de C# 3.0 estipulan que toda expresión de consulta que aparece en el código fuente antes de ser compilada es transformada (**reescrita**) de una manera mecánica en una secuencia de llamadas a métodos con nombres y firmas predeterminadas de antemano, que reciben el nombre de **operadores de consulta** (*query operators*). Las expresiones

que forman parte de cada una de las cláusulas que conforman la expresión de consulta también se reescriben para adecuarlas a los requerimientos de esos métodos predeterminados. Antes de compilarla propiamente, el compilador traducirá la expresión de consulta de nuestro ejemplo anterior en lo siguiente:

```
var mayores20 = Datos.Personas
                .Where(h => h.Edad > 20)
                .OrderBy(h => h.Nombre)
                .Select(h => new
                {
                    Nombre = h.Nombre.ToUpper(),
                    Edad = h.Edad
                });
```

La cláusula **where** de la expresión de consulta se transforma en una llamada a un método llamado `Where()`. Para pasarla a ese método, la expresión que acompañaba a **where**:

```
h.Edad > 20
```

es transformada en una expresión lambda que a partir de una `Persona h` produce verdadero o falso:

```
h => h.Edad > 20
```

Espero que le encuentre a esto toda la lógica del mundo: si el método `Where()` va a tener un carácter general, o sea, si va a ser capaz de funcionar para cualquier condición que un programador pudiera necesitar, lo lógico es que reciba como parámetro un **delegado** que apunte a un método booleano que compruebe la condición a cumplir. Las expresiones lambda son, precisamente, una manera más funcional de especificar delegados anónimos.

Una vez comprendido lo anterior, la siguiente pregunta es: ¿cuál es (o debe ser) la firma de este método `Where()` en el que se traduce la cláusula **where**? ¿Dónde debe estar situado? Observe que después del primer paso de reescritura de nuestra expresión de consulta de ejemplo la cosa quedaría así:

```
var mayores20 = Datos.Personas.Where(h => h.Edad > 20)
```

Para que esa llamada sea válida, `Where()` debe ser: a) un método de instancia de la clase a la que pertenece `Personas` o b) un método de una interfaz implementada por la clase a la que pertenece `Personas` (y dado que hemos partido de que los objetos que pueden servir como origen para las consultas integradas deben implementar `IEnumerable<T>`, esa interfaz sería una buena candidata para ser extendida con métodos como `Where()`, etc.).

Cualquiera de las dos vías antes mencionadas podría en principio valer, pero los creadores de C# 3.0 creyeron que llevarían a arquitecturas menos abiertas y extensibles. Es en este punto de la representación en que entran en escena los **métodos extensores**. Teniendo los métodos extensores a nuestra disposición,

`Where()`, al igual que `OrderBy()`, `Select()` y demás actores, podrían ser también métodos extensores de `IEnumerable<T>`, que potencialmente podrían estar definidos en cualquier clase estática que esté en ámbito en el momento en que la expresión de consulta se compile.

Para comprobar la teoría anterior, haga un pequeño experimento: en el código fuente del ejemplo comente la línea en la parte superior del fichero que dice:

```
using System.Linq;
```

Verá que el programa deja de compilar (analice en detalle el mensaje de error: "No se puede encontrar una implementación del patrón de consultas para la clase `List<Persona>`. 'Where' no encontrado."). La razón para ello es que al comentar la sentencia ***using***, se ha privado al compilador de las definiciones de los métodos extensores `Where()`, `OrderBy()`, `Select()`, etc. (que se conocen genéricamente como **operadores de consulta**) en los que se traducen las diferentes cláusulas de la consulta integrada. Las implementaciones predeterminadas de estos operadores están alojadas en una clase estática llamada muy apropiadamente `System.Linq.Enumerable`, e implementada en el ensamblado `System.Core.dll`, al que hacen referencia automáticamente todos los proyectos para.NET Framework 4.1 creados con Visual Studio 2010).

1.4.- La (no) semántica de los operadores de consulta

Como habrá sacado ya en claro, las expresiones de consulta son puro "azúcar sintáctico". En la sección anterior hemos visto cómo estas expresiones se traducen mecánicamente, siguiendo un conjunto de reglas predefinidas en la especificación del lenguaje, en una secuencia de llamadas a métodos. También hemos visto cómo al eliminar una importación de espacio de nombres el código que contiene una expresión de consulta deja de compilar. En ese momento nos faltó enfatizar que si ponemos en ámbito **otro** espacio de nombres que contenga clases estáticas con las definiciones apropiadas de esos métodos extensores, la expresión de consulta volverá a compilar sin problemas, y utilizará en su ejecución el nuevo conjunto de métodos-operadores. En esto consiste la **arquitectura abierta** de LINQ: C# no define una semántica específica para los operadores que implementan las expresiones de consulta, y cualquiera puede crear una o más clases con implementaciones a medida de los operadores de consulta para colecciones genéricas o específicas y, poniéndola en ámbito, "enchufarla" al sistema para que sea utilizada cuando se compilen las consultas integradas sobre secuencias de esos tipos. Precisamente ésta es la vía a través de la cual se integran en el lenguaje las extensiones predefinidas de LINQ, como LINQ to XML o LINQ to Entities, y también la vía a través de la cual terceros fabricantes pueden desarrollar proveedores propietarios.

1.5.- Resolución de llamadas a operadores

Aunque el desarrollo de extensiones (proveedores) de LINQ va más allá de los objetivos de este Apéndice, veamos solo un ejemplo que nos aclare otro tema relacionado: la resolución de las llamadas. Suponga que queremos que nuestras consultas integradas sobre secuencias del tipo `Persona` se comporten de manera tal, que cuando se pregunte qué personas cumplen con una condición `p` cualquiera se devuelvan solo las personas de sexo femenino que cumplen con dicha condición (lo cual no sería correcto, pero sin duda elevaría el "coeficiente de belleza" de nuestros conjuntos de resultados :-). Podríamos definir (para simplificar, en la propia clase `Program`), el siguiente método `Where()`, que opera sobre un tipo genérico cerrado (`IEnumerable<Persona>`):

```
public static IEnumerable<Persona> Where(
            this IEnumerable<Persona> origen,
            Func<Persona, bool> filtro)
        {
            foreach (Persona p in origen)
                if (filtro(p) && p.Sexo == SexoPersona.Mujer)
                {
                    yield return p;
                }
        }
```

Si compila y ejecuta ahora la consulta que selecciona los mayores de 20 años, podrá comprobar que se hará uso de nuestro método recién creado y no del operador de consulta estándar, y que por tanto en el resultado aparecerán solo chicas. Debido a que la consulta en cuestión opera sobre un objeto de tipo `IEnumerable<Persona>`, nuestro método recibe preferencia sobre el de la clase `System.Linq.Enumerable`. ¿Y qué ocurre con `OrderBy()`, `Select()` y demás? Pues que nosotros no hemos definido esos métodos, pero `IEnumerable<Persona>` es compatible con `IEnumerable<T>` y por lo tanto se utilizarán las implementaciones de esos métodos situadas en la librería de clases base.

Una última cuestión a tener en cuenta: ¿y si hubiéramos definido el método `Where()` para que operara sobre `IEnumerable<T>`, al igual que en la implementación predeterminada? Pues se utilizaría igualmente nuestra versión, por estar situada en el mismo espacio de nombres que la clase en la que se ejecuta la consulta. El algoritmo de resolución de llamadas del compilador va buscando en los espacios de nombres de nuestro código de dentro hacia afuera, y solo si no encuentra nada por esa vía utiliza los métodos que encuentre en clases estáticas pertenecientes a otros espacios de nombres en ámbito.

1.6.- Ejecución diferida

Las implementaciones predeterminadas de los operadores de consulta funcionan, al estilo de cómo lo hemos hecho en el código anterior, mediante iteradores, lo cual se

refleja a través del uso de la sentencia `yield return`. Este hecho da lugar a la particularidad de que la asignación de una expresión de consulta como:

```
var mayores20 = from h in Datos.Personas
                where h.Edad > 20
                orderby h.Nombre
                select new
                {
                    Nombre = h.Nombre.ToUpper(),
                    Edad = h.Edad
                };
```

No hace más que preparar los objetos enumeradores necesarios; el resultado de una consulta no se obtendrá realmente hasta el momento en que se produzca la iteración sobre ella. Esta evaluación **bajo demanda**, **perezosa** o **diferida** constituye el comportamiento por defecto de LINQ, y un posible efecto colateral de ella que siempre debemos tener en cuenta es el hecho de que dos evaluaciones sucesivas de una misma consulta produzcan resultados diferentes si entre medias se producen cambios en la fuente de información subyacente. A pesar de ello, la ejecución diferida es, en la mayor parte de los casos, la opción más conveniente. No obstante, puede que en cierto momento se desee "cachear" completamente en memoria el resultado de una consulta para su posterior reutilización. Con este objetivo se ofrecen los operadores de consulta estándar `ToArray()`, `ToList()`, `ToLookup()` y `ToDictionary()`. Por ejemplo, podemos obtener de una vez los resultados de la consulta anterior mediante la sentencia:

```
var listaMayores20 = mayores20.ToList();
```

1.7.- Los operadores de consulta estándar

Bueno, ya está claro que potencialmente podemos hacer que los métodos que implementan los operadores de consulta hagan lo que nos dé la gana, siempre que cumplan con las firmas que exige el compilador; pero se supone que les asociemos una funcionalidad que cumpla con lo que en general se espera de ellos. El método `Where()`, por ejemplo, se supone que filtre la secuencia de entrada, dejando en la salida solo los elementos que satisfagan la condición especificada. `OrderBy()`, por su parte, debe recoger la secuencia de entrada y producir otra que contenga los mismos elementos que la original, pero ordenados ascendentemente según un cierto criterio.

Continuando con `Where()`, a estas alturas conocemos en su totalidad la firma del método, al menos en su sobrecarga principal. Suponiendo que se implementa como método extensor, recibe como primer argumento la secuencia de entrada (marcada con `this`) y como segundo parámetro un delegado a una función que recibe un `T` y devuelve un `bool`. El tipo del valor de retorno es `IEnumerable<T>`, como se desprende de nuestro último ejemplo de código; no es difícil darse cuenta de ello, teniendo en cuenta que el resultado que `Where()` produce servirá como entrada a `OrderBy()`, `Select()` o algún otro de los métodos en la cascada de llamadas que se obtiene como resultado de la reescritura.

La sobrecarga principal del operador de consulta estándar `Where()` está implementada así:

```
// *** Operador Where() alternativo
        public static IEnumerable<T> Where<T>(
            this IEnumerable<T> origen, Func<T, bool> filtro)
        {
            if (origen == null || filtro == null)
                throw new ArgumentNullException();
            foreach (T t in origen)
                if (filtro(t))
                {
                    yield return t;
                }
        }
```

El método comprueba inicialmente la validez de los argumentos de entrada. Acto seguido, entra en un bucle que recorre la secuencia de entrada, llamando para cada uno de los elementos de ésta al predicado para comprobar si el elemento cumple con la condición o no. Solo en el caso de que el elemento cumpla con la condición, lo produce en su secuencia de salida. En nuestro ejemplo de los mayores de 20 años, esa secuencia de salida será a su vez la secuencia de entrada para el operador `OrderBy()`.

Como otro ejemplo, vea cómo se implementa el operador `Select()`:

```
// *** Operador Select() alternativo
public static IEnumerable<V> Select<T, V>(
    this IEnumerable<T> origen, Func<T, V> selector)
{
    if (origen == null || selector == null)
        throw new ArgumentNullException();
    foreach (T t in origen)
        yield return selector(t);
}
```

Aquí el tipo de los elementos de la secuencia resultante viene dado por el tipo de retorno de la expresión de selección o transformación que se utilice.

La implementación predeterminada (en la clase `System.Linq.Enumerable` de `System.Core.dll`) de un conjunto de métodos extensores entre los que se incluyen `Where()`, `Select()`, `OrderBy()` y otros más, que pueden utilizarse para ejecutar consultas integradas contra cualquier secuencia enumerable de objetos en memoria se conoce como **LINQ to Objects**, y a esos métodos se les conoce como **operadores de consulta estándar** (*standard query operators*).

1.8.- El patrón de expresiones de consulta

No todos los operadores de consulta estándar tienen un reflejo en la sintaxis del lenguaje. Por ejemplo, existe un operador llamado `Reverse()` que produce los elementos de su secuencia de origen en orden inverso (desde el último hasta el

primero). Sin embargo, no existe un "mapeado" sintáctico en el lenguaje C# para ese operador. Cuando lo necesitemos, tendremos que usarlo con la notación habitual de llamada a método:

```
var alReves = mayores20.Reverse()
```

Ni siquiera todas las sobrecargas de un mismo operador de consulta estándar tienen reflejo en la sintaxis de las expresiones de consulta de C#, sino solo algunas. Por ejemplo, el operador `Where()` ofrece dos variantes, pero solo una de ellas, la que hemos presentado anteriormente, es la que se utiliza para la reescritura de expresiones de consulta.

El subconjunto de los operadores de consulta estándar de C# 3.0 de los cuales la sintaxis de expresiones de consulta tiene una dependencia directa (y que, por tanto, cualquier fabricante de extensiones de LINQ debería soportar) da lugar a lo que se conoce como patrón de expresiones de consulta o **patrón LINQ**: una especificación del conjunto de métodos (subconjunto del conjunto de operadores de consulta estándar) de los que se debe disponer para garantizar un soporte completo para las consultas integradas.

1.9.- Sintaxis de las expresiones de consulta

La sintaxis completa de las expresiones de consulta es la siguiente:

```
<expr. consulta> ::= <cláusula from> <cuerpo consulta>

<cláusula from> ::=
        from <elemento> in <expr.origen>

<cuerpo consulta> ::=
        <cláusula cuerpo consulta>*
        <cláusula final consulta>
        <continuación>?

<cláusula cuerpo consulta> ::=
       (<cláusula from>
      | <cláusula join>
      | <cláusula join-into>
      | <cláusula let>
      | <cláusula where>
      | <cláusula orderby>)

<cláusula let> ::=
        let <elemento> = <expr.selección>

<cláusula where> ::=
        where <expr.filtro>
```

```
<cláusula join> ::=
        join <elemento> in <expr.origen>
        on <expr.clave> equals <expr.clave>

<cláusula join-into> ::=
        join <elemento> in <expr.origen>
        on <expr.clave> equals <expr.clave>
        into <elemento>

<cláusula orderby> ::=
        orderby <ordenaciones>

<ordenaciones> ::=
        <ordenacion>
      | <ordenaciones> , <ordenación>

<ordenación> :=
        <expr.clave> (ascending | descending)?

<cláusula final consulta> ::=
       (<cláusula select> | <cláusula groupby>)

<cláusula select> ::=
        select <expr.selección>

<cláusula groupby> ::=
        group <expr.selección> by <expr.clave>

<continuación> ::=
        into <elemento> <cuerpo consulta>

Meta-lenguaje:
  * - cero o más veces
  ( ... | ... ) - alternativa
  ? - elemento opcional
```

Básicamente, una expresión de consulta siempre comienza por una cláusula `from`, en la que especifica la colección de origen sobre la que se ejecutará la consulta. A continuación pueden venir una o más cláusulas `from`, `join`, `let`, `where` u `orderby`, para terminar con una sentencia `select` o `group by`. Opcionalmente, al final de la expresión puede aparecer una cláusula de continuación, que comienza con la palabra reservada `into` y va seguida del cuerpo de otra consulta. Recuerde que todas las palabras clave que se utilizan aquí son contextuales – tienen un significado especial solo dentro de las expresiones de consulta.

En una próxima sección mostraremos ejemplos de utilización de los diferentes elementos sintácticos de las expresiones de consulta.

1.10.-Tabla de operadores de consulta estándar

La tabla que se presenta a continuación lista los operadores de consulta estándar disponibles, agrupados por categorías. Hemos destacado al principio los operadores "básicos" para los cuales la sintaxis de las expresiones de consulta ofrece una cláusula especial (`where`, `orderby`, `select`, `group`, `join`) y algunas de cuyas sobrecargas forman parte del ya mencionado **patrón de expresiones de consulta**, que se define exactamente en el documento de especificación de C# 3.0. Para el resto de los operadores estándar no existe soporte lingüístico directo en C# 3.0, y para utilizarlos se deberá hacer uso de la sintaxis explícita de llamada a método. Observe que, si bien los operadores básicos y muchos de los no básicos producen como resultado otra secuencia, entre los demás operadores hay varios que producen como resultado valores escalares, lo que significa que deberán situarse siempre al final de la "cadena de producción".

Tabla 1.- Operadores de consulta estándar

Operadores del patrón LINQ	
Where()	Filtrado de la secuencia original en base a un predicado lógico.
Select()/SelectMany()	Proyección de la secuencia original en otra en base a una función de transformación.
OrderBy()/ThenBy()	Ordenación ascendente de la secuencia original en base a una función de cálculo de la clave de ordenación.
OrderByDescending()/ ThenByDescending()	Ordenación descendente de la secuencia original en base a una función de cálculo de clave de ordenación.
GroupBy()	Creación de grupos a partir de la secuencia original en base a una función de cálculo de la clave de agrupación.
Join()	Encuentro interno de la secuencia original y una segunda secuencia en base a funciones de cálculo de la clave de encuentro para cada una de las secuencias.
GroupJoin()	Encuentro agrupado de la secuencia original y una segunda secuencia en base a funciones de cálculo de la clave de encuentro para cada una de las secuencias.
Operadores de partición	
Take()	Selección de elementos de la secuencia original situados antes de la posición especificada.
Skip()	Selección de elementos de la secuencia original situados después de la posición especificada.

TakeWhile()	Selección de elementos de la secuencia original mientras un predicado es satisfecho.
SkipWhile()	Selección de elementos de la secuencia original situados a partir de que un predicado deja de cumplirse.
Operadores conjuntuales	
Distinct()	Selección de elementos únicos de la secuencia original.
Union()	Unión de la secuencia original y una segunda secuencia.
Intersect()	Intersección de la secuencia original y una segunda secuencia.
Except()	Diferencia de la secuencia original y una segunda secuencia.
Operadores de conversión	
ToArray()	Creación de un array a partir de los elementos de la secuencia original.
ToList()	Creación de una lista genérica (`List<T>`) a partir de los elementos de la secuencia original.
ToDictionary()	Creación de un diccionario de pares clave/valor (`Dictionary<K, V>`) a partir de la secuencia original y de funciones de cálculo de la clave y el valor.
ToLookup()	Creación de un diccionario de pares clave/secuencia de elementos con ese valor de clave (`Lookup<K, V>`) a partir de la secuencia original y de funciones de cálculo de la clave y el valor.
AsEnumerable()	Cambio del tipo de la secuencia original a `IEnumerable<T>`.
Cast<T>()	Conversión del tipo de los elementos de la secuencia original a `T`.
OfType<T>()	Filtrado de los elementos de la secuencia original que son del tipo `T`.
Operadores de generación de secuencias	
Range()	Generación de una secuencia formada por `n` números enteros consecutivos a partir de un cierto valor `m`.
Repeat<T>()	Generación de una secuencia en la que se repite `n` veces un elemento de tipo `T`.
Empty<T>()	Generación de una secuencia vacía de elementos de tipo `T`.
Operadores de transformación de secuencias	
Concat()	Concatenación de dos secuencias.
Reverse()	Inversión de una secuencia.

Cuantificadores	
Any()	Cuantificador existencial: devuelve `true` si alguno de los elementos de la secuencia original satisface un predicado lógico, o `false` en caso contrario.
All()	Cuantificador universal: devuelve `true` si todos los elementos de la secuencia original satisfacen un predicado lógico, o `false` en caso contrario.
Contains()	Comprueba la existencia de un elemento dado dentro de la secuencia original.
SequenceEqual()	Comprueba si la secuencia original y una segunda secuencia son iguales.
Elementos	
First()	Devuelve el primer elemento de la secuencia original, o el primer elemento para el que se cumple una condición especificada.
FirstOrDefault()	Devuelve el primer elemento de la secuencia original, o el primer elemento para el que se cumple una condición especificada. Si tal elemento no existe, devuelve el valor predeterminado del tipo de los elementos de la secuencia original.
Last()	Devuelve el último elemento de la secuencia original, o el último elemento para el que se cumple una condición especificada.
LastOrDefault()	Devuelve el último elemento de la secuencia original, o el último elemento para el que se cumple una condición especificada. Si tal elemento no existe, devuelve el valor predeterminado del tipo de los elementos de la secuencia original.
Single()	Devuelve el único elemento de la secuencia original, o el único elemento para el que se cumple una condición especificada.
SingleOrDefault()	Devuelve el único elemento de la secuencia original, o el único elemento para el que se cumple una condición especificada. Si tal elemento no existe, devuelve el valor predeterminado del tipo de los elementos de la secuencia original.
ElementAt()	Devuelve el elemento de la secuencia original situado en la posición especificada.
ElementAtOrDefault()	Devuelve el elemento de la secuencia original situado en la posición especificada. Si tal elemento no existe, devuelve el valor

	predeterminado del tipo de los elementos de la secuencia original.
DefaultIfEmpty()	Devuelve la misma secuencia original, o una secuencia formada por un valor predeterminado si la secuencia original es vacía.
Operadores de consolidación (agregados)	
Count() / LongCount()	Devuelve la cantidad de elementos en la secuencia original, o la cantidad de elementos que satisfacen un predicado lógico especificado.
Max() / Min()	Devuelve el mayor (menor) de los elementos de la secuencia original.
Sum()	Devuelve la suma de los elementos de la secuencia (numérica) original.
Average()	Devuelve la media de los elementos de la secuencia (numérica) original.
Aggregate()	Devuelve el resultado de aplicar una función de consolidación dada a los elementos de la secuencia original.

1.11.-Algunos ejemplos

El objetivo de esta sección es presentar algunos ejemplos de las cosas que pueden lograrse utilizando expresiones de consulta.

Ejemplos básicos

Partiendo del hecho de que cualquier objeto que implemente `IEnumerable<T>` puede servir como origen para una consulta integrada, y que objetos tan habituales como los arrays, las colecciones genéricas e incluso las cadenas de caracteres (lo que nos permite enumerar los caracteres que las componen) implementan esta interfaz, queda claro que podremos aplicar las consultas integradas en una gran cantidad de situaciones cotidianas para las que anteriormente utilizábamos bucles, contadores y otras técnicas varias.

A continuación se muestran algunos ejemplos de expresiones de consulta aplicadas a cadenas y arrays:

```
string s = "Hasta la vista, baby";

// produce en orden alfabético las vocales de la cadena 's'
var s1 = from c in s
         where "AEIOU".Contains(char.ToUpper(c))
         orderby c
         select c;

// cuenta los espacios en la cadena 's'
// observe que no hay sintaxis para el operador Count()
int n1 = (from c in s
```

```
                where c == ' '
                select c).Count();

// produce las palabras diferentes en una oración
// string.Split() produce un array de cadenas
// Distinct() tampoco tiene "sintaxis endulzada"
var n2 = (from w in s.Split(new char[] { ' ', '\t', '\n' },
                   StringSplitOptions.RemoveEmptyEntries)
                orderby w.ToLower()
                select w.ToLower()).Distinct();

int[] arr = { 2, 4, 3, 7, 25, 9, 6 };

// produce una secuencia con los pares en arr
var pares = from n in arr
            where n % 2 == 0
            select n;
// también pudo ser:
var pares2 = arr1.Where(n => n % 2 == 0);

// produce la suma de los números de la secuencia
int suma = arr.Sum();
// lo mismo que:
int suma2 = (from n in arr select n).Sum();

// produce los números de la secuencia, incrementados en 1
var otra = from n in arr
           select n + 1;
// lo mismo que:
var otra2 = arr.Select(n => n + 1);
```

Por último, veamos algunos ejemplos sobre la lista de personas:

```
// nombres que empiezan por 'D'
var hijos = from h in Datos.Personas
            where h.Nombre.StartsWith("D")
            select h.Nombre;

// lista ordenada, primero por sexos
// luego por edad en orden descendente
var orden = from h in Datos.Personas
            orderby h.Sexo, h.Edad descending
            select h;
```

Productos cartesianos

En el mundo de las bases de datos relacionales, un **producto cartesiano** de dos tablas no es más que el conjunto de filas resultante de combinar cada fila de la primera tabla con cada fila de la segunda. El mismo concepto se puede aplicar aquí a la combinación de dos secuencias: si se colocan dos cláusulas `from` (que actúan como generadoras) seguidas, para cada elemento de la primera secuencia se producirán todos los elementos de la segunda. Por ejemplo:

```
/* PRODUCTO CARTESIANO */
var pc1 = from pa in Datos.Paises
          from pe in Datos.Personas
          select new {
              pa.Nombre,
              NombrePersona = pe.Nombre
          };
```

Los productos cartesianos se implementan mediante llamadas al operador de consulta estándar `SelectMany()`, que es el encargado de producir una secuencia en la que se combina cada uno de los elementos de la primera secuencia con cada uno de los elementos de la segunda. Veremos con más detalle este operador en el siguiente capítulo.

El principal peligro de los productos cartesianos es la explosión combinatoria de resultados que pueden provocar. Por ello en general es recomendable evitar los productos cartesianos siempre que sea posible, posiblemente aplicando alguna de las técnicas que se describen a continuación.

Restricción de productos y optimización de consultas

Suponga que queremos obtener una lista de parejas de personas en las que el primer elemento es un chico y el segundo una chica. Un primer intento podría ser:

```
var pc2 = from p1 in Datos.Personas
          from p2 in Datos.Personas
          where p1.Sexo == SexoPersona.Varón &&
                p2.Sexo == SexoPersona.Mujer
          select new { El = p1.Nombre, Ella = p2.Nombre };
```

El anterior es un ejemplo de lo que podríamos llamar **producto cartesiano restringido**: un producto cartesiano al que se adjuntan condiciones de filtro que reducen el tamaño de la secuencia resultante.

Si analiza con detenimiento la consulta anterior, le parecerá bastante obvio que la siguiente es una mejor opción en relación con el rendimiento, porque los elementos de la primera secuencia que a fin de cuentas no van a servir son eliminados antes en la "tubería" de operadores de consulta que se ejecutan:

```
var pc3 = from p1 in Datos.Personas
          where p1.Sexo == SexoPersona.Varón
           from p2 in Datos.Personas
           where p2.Sexo == SexoPersona.Mujer
           select new { El = p1.Nombre, Ella = p2.Nombre };
```

Aunque el estudio de técnicas específicas de optimización de consultas LINQ queda fuera de los objetivos de este Apéndice, no está de más comentar este hecho para que lo tenga en cuenta al programar sus consultas integradas. Otra cuestión totalmente diferente es que un compilador "inteligente" podría transformar de forma transparente

al programador la primera expresión en la segunda; seguramente futuras versiones del compilador de C# lo harán, pero no la actual.

Encuentros

Los encuentros son otra de las construcciones típicas de los lenguajes relacionales como SQL que han sido incorporadas a las expresiones de consulta de C# 3.0. Un **encuentro** es básicamente un producto cartesiano sobre dos secuencias limitado a las tuplas `(t1, t2)` en las que el valor de cierta expresión aplicada al elemento de la primera secuencia `t1` **es igual** al valor de otra expresión aplicada al elemento de la segunda secuencia `t2`. Básicamente, se trata de limitar drásticamente las combinaciones que produciría un producto cartesiano completo, manteniendo únicamente los elementos de las secuencias que "casan" de acuerdo con cierto criterio compartido.

Por ejemplo, la consulta que "combina" los nombres de personas y de países presentada anteriormente quedaría mucho mejor así:

```
/* ENCUENTRO */
var enc1 = from pa in Datos.Paises
           join pe in Datos.Personas
               on pa.Codigo equals pe.CodigoPaisNac
           select new {
               pa.Nombre,
               NombrePersona = pe.Nombre
           };
```

La condición de encuentro entre las dos secuencias consiste en un selector de clave para la secuencia externa, la palabra clave contextual **`equals`** y otro selector de clave para la secuencia interna. Los selectores de claves que se utilizan para la comparación de campos pueden ser cualquier expresión que se obtenga a partir del identificador que representa al elemento de la secuencia correspondiente.

El mismo resultado puedo haberse obtenido mediante un producto cartesiano restringido, pero el rendimiento del encuentro es muy superior. El método extensor `Join()` (que es llamado para ejecutar los encuentros) se orienta a la utilización de tablas hash, de un modo similar a como se utilizan los índices de las tablas en el mundo de las bases de datos.

Grupos

La sintaxis de las expresiones de consulta ofrece también soporte para la organización de los elementos de una secuencia en grupos según los diferentes valores de una clave de agrupación que se calcula para cada uno de los elementos. Por ejemplo, la siguente sentencia:

```
/* AGRUPACION */
var gruposSexo =
    from h in Datos.Personas
    group new { h.Nombre, h.Edad } by h.Sexo;
```

agrupa los elementos de la secuencia original según los diferentes valores de la expresión `h.Sexo`. En este caso, se obtendrá una secuencia de dos elementos, que serán

a su vez secuencias: la primera, con objetos de un tipo anónimo que incluye los datos solicitados (nombre y edad) de todas las chicas (objetos para los que el valor de la clave de agrupación es `SexoPersona.Mujer`); la segunda, con los datos de todos los chicos, para los que el valor de `h.Sexo` es igual a `SexoPersona.Varón`.

C# 3.0 traduce la expresión de consulta anterior en una llamada al operador estándar `GroupBy()`; el resultado es una secuencia cada uno de cuyos elementos son a su vez otras secuencias internas asociadas a cada grupo, que implementan una interfaz llamada `IGrouping<TKey, TElmt>`, heredera de `IEnumerable<TElemt>`. Básicamente, esta interfaz añade una propiedad de solo lectura `Key`, del tipo de la clave de agrupación. El siguiente bucle muestra la estructura del resultado de la consulta:

```
foreach (var hh in gruposSexo)
{
    // la clave del grupo
    Console.WriteLine(hh.Key);
    // los elementos del grupo
    foreach (var hhh in hh)
        Console.WriteLine(" - " + hhh.Nombre +
                          " (" + hhh.Edad + ")");
}
```

El ejemplo que se presenta a continuación involucra nuestras dos "tablas" de personas y países. La siguiente sentencia permite agrupar las personas según sus países de nacimiento:

```
Console.WriteLine("GRUPOS POR PAISES");
var gruposPaises =
    from pa in Datos.Paises
    join pe in Datos.Personas
        on pa.Codigo equals pe.CodigoPaisNac
    group new { pe.Nombre, pe.Edad }
        by pa.Nombre;

foreach (var hh in gruposPaises)
{
    // el valor de la clave
    Console.WriteLine(hh.Key);
    // los elementos del grupo
    foreach (var hhh in hh)
        Console.WriteLine(" - " + hhh.Nombre +
                          " (" + hhh.Edad + ")");
}
```

Observe la similitud entre esta última expresión de consulta y la que presentamos en la sección dedicada a los encuentros; la diferencia está en la presencia de la cláusula `group...by` en lugar de `select`. Precisamente estas dos cláusulas son las "cláusulas finales" en la sintaxis de las expresiones de consulta.

Continuaciones

Si ejecuta la agrupación anterior, verá que los diferentes grupos aparecen en la secuencia resultante en el mismo orden en que aparecen los países en la secuencia

original. ¿Y si quisiéramos obtener los grupos en orden alfabético de los países? Podríamos acudir a la sintaxis explícita:

```
var gruposPaises2 =
                (from pa in Datos.Paises
                join pe in Datos.Personas
                    on pa.Codigo equals pe.CodigoPaisNac
                group new { pe.Nombre, pe.Edad }
                    by pa.Nombre).OrderBy(g => g.Key);
```

Para expresar este tipo de situaciones el lenguaje prevee un mecanismo mejor: las continuaciones, que permiten establecer una "cascada" de consultas, en las que los resultados de una consulta se utilizan como secuencia de entrada para la consulta subsiguiente. Para ello se utiliza la cláusula `into`:

```
var gruposPaises3 =
                from pa in Datos.Paises
                join pe in Datos.Personas
                    on pa.Codigo equals pe.CodigoPaisNac
                group new { pe.Nombre, pe.Edad } by pa.Nombre
                    into tmp
                    orderby tmp.Key
                    select tmp;
```

En la práctica, las continuaciones son especialmente útiles para procesar los resultados producidos por una cláusula `group...by`. Observe este otro ejemplo, parecido al anterior:

```
var resumenPaises =
    from pa in Datos.Paises
    join pe in Datos.Personas
        on pa.Codigo equals pe.CodigoPaisNac
    group new { pe.Nombre, pe.Edad } by pa.Nombre
        into tmp
        orderby tmp.Count() descending
        select new {
            Nombre = tmp.Key,
            Cantidad = tmp.Count()
        };
```

Esta consulta produce una secuencia ordenada de objetos con dos propiedades: el nombre del país y la cantidad de personas nacidas en ese país.

Encuentros agrupados

La segunda y más importante aplicación de la cláusula `into` tiene como objetivo implementar lo que se conoce como **encuentros agrupados** (*grouped joins*). Se trata de un tipo de encuentro que no tiene equivalente directo en el mundo de las bases de datos relacionales, y que en lugar de producir la típica secuencia de parejas de un encuentro normal produce una secuencia en la que cada elemento de la primera secuencia se aparea con el **grupo de elementos** de la segunda cuyos valores de clave

de comparación coinciden con el valor de la clave de comparación del elemento de la secuencia externa. Una construcción `join...into` se traduce en una llamada al operador estándar `GroupJoin()`, que se basa, al igual que `Join()`, en el uso de tablas hash.

Por ejemplo, una manera más concisa de haber obtenido una lista de países con las cantidades de personas de ese país similar a la del epígrafe anterior habría sido:

```
var resumenPaises2 =
    from pa in Datos.Paises
    orderby pa.Nombre
    join pe in Datos.Personas
        on pa.Codigo equals pe.CodigoPaisNac
    into gp
    select new {
        Pais = pa.Nombre,
        Cantidad = gp.Count()
    };
```

La traducción de esta expresión de consulta sería:

```
var resumenPaises3 =
    Paises.OrderBy(pa => pa.Nombre).
    GroupJoin(Generación,
        pa => pa.Codigo,
        pe => pe.CodigoPaisNac,
        (p, gp) => new { Pais = p.Nombre,
                         Cantidad = gp.Count() });
```

Observe que, a diferencia de aquella consulta, en el resultado de ésta se incluirán también los países en los cuales no ha nacido nadie.

La cláusula let

Por último, la cláusula `let` sirve como un mecanismo de conveniencia muy útil en los casos en que necesitamos asignar un nombre a un resultado intermedio para aprovecharlo más adelante o ejecutar una consulta "subsidiaria" (subconsulta) cuyo resultado necesitamos dentro de otra consulta.

Por ejemplo, suponga que tenemos una secuencia de números enteros, y queremos agruparlos según su última cifra. Para cada número de la secuencia necesitaremos obtener esa última cifra, dado que es en base a éste que se va a agrupar. Podríamos hacerlo así:

```
var let1 = from n in arr
           let terminacion = n % 10
           orderby terminacion
           group n by terminacion;
```

El compilador traduce una cláusula `let` inyectando en el código fuente lo que se conoce como un **identificador transparente**. En la práctica, ello se traduce en una llamada al operador `Select()` que incorpora los valores de "la variable" a los elementos

de la secuencia de salida como una nueva propiedad adicional. La sentencia anterior se traducirá a:

```
var let2 = arr.
                Select(n => new { n, terminacion = n % 10 }).
                OrderBy(x => x.terminacion).
                GroupBy(x => x.terminacion, x => x.n);
```

Como ejemplo de utilización de `let` para expresar subconsultas, suponga que queremos obtener las personas cuya edad es mayor o igual que la media de edad del conjunto. En principio, necesitaríamos ejecutar dos consultas: una inicial para calcular la media, y luego una segunda para obtener los elementos cuya edad supere esa media anteriormente calculada. Con la ayuda de `let`, podríamos expresarlo todo de una vez así:

```
var let3 = from p in Datos.Personas
           let media = (Datos.Personas.Average(pp => pp.Edad))
             where p.Edad >= media
             select p.Nombre;
```

1.12.-Extensiones de LINQ

Como ya hemos comentado anteriormente, los operadores de consulta no tienen una semántica predeterminada, sino que ésta se "enchufa" en tiempo de compilación en función del tipo de la fuente de datos a la que se aplica la consulta y de los conjuntos de métodos extensores que estén en ámbito en ese momento. La siguiente tabla muestra los ensamblados que contienen las clases extensoras y los espacios de nombres a los que estas clases pertenecen para cada una de las extensiones de LINQ disponibles "de serie" en .NET Framework 3.5:

Tabla 2.- Extensiones de LINQ

Tecnología	Ensamblado	Espacio de nombres
Proveedores locales		
LINQ to Objects	System.Core.dll	System.Linq
LINQ to XML	System.Xml.Linq.dll	System.Xml.Linq
LINQ to DataSets	System.Data.DataSetExtensions.dll	System.Data
Proveedores remotos		
LINQ to SQL	System.Data.Linq.dll	System.Data.Linq
LINQ to Entities	System.Data.Entity.dll	System.Data.Objects y otros

Como puede observarse, los proveedores LINQ pueden clasificarse en dos categorías muy diferentes: los proveedores locales y los remotos. La siguiente tabla resume brevemente las diferencias fundamentales entre ellos.

Tabla 3.- Diferencias entre proveedores locales y remotos

	Proveedores locales	Proveedores remotos
Interfaz	IEnumerable<T>	IQueryable<T>
Ejecución	Local, en memoria	Generalmente remota
Implementación	Delegados anónimos	Árboles de expresiones
Ejemplos del libro "C# 3.0 y LINQ"	LINQ to Pipes, LoggingLINQ	LINQ to TFS
Otros ejemplos	Parallel LINQ	LINQ to Amazon, LINQ to LDAP

Se denominan **proveedores locales** aquellos que operan sobre fuentes de datos disponibles en la memoria operativa del ordenador en el que la aplicación se ejecuta. Además de LINQ to Objects, en esta categoría caen LINQ to XML (que permite consultar documentos XML cargados, como árboles de nodos, en memoria) y LINQ to DataSets (orientado a la consulta de conjuntos de datos en memoria de ADO.NET). Debido a que en el fondo operan sobre secuencias (que implementan `IEnumerable<T>`), tales proveedores generalmente se apoyan en las implementaciones de los operadores de consulta estándar que ofrece LINQ to Objects. Por ejemplo, si revisa con el Examinador de Objetos el ensamblado `System.Xml.Linq.dll`, que contiene la implementación de LINQ to XML, no encontrará ninguna clase que tenga métodos extensores llamados `Where()`, `Select()`, etc. LINQ to XML se apoya en LINQ to Objects; en consecuencia, para programar una consulta integrada contra un documento de LINQ to XML no solo será necesario importar el espacio de nombres `System.Xml.Linq`, sino también `System.Linq`. Por otra parte, estos proveedores ofrecen operadores de consulta específicos, que siempre deberán expresarse con notación funcional; por ejemplo, LINQ to XML ofrece varios operadores (como `Elements()`) que emulan las posibilidades de selección del lenguaje XPath.

Mucho más poderosos e interesantes son los **proveedores remotos**, categoría a la que pertenece LINQ to Entities, y a los que dedicaremos las siguientes secciones.

1.13.-La interfaz IQueryable<T>

Tan pronto se pensó en implementar una extensión de LINQ para consultar bases de datos relacionales, se hizo evidente que el mecanismo basado en el recorrido de secuencias en memoria no era el más adecuado. Cualquier implementación de los operadores de consulta que estuviera basada en la navegación de cursores de cliente, incluso utilizando evaluación demorada, sería desastrosa desde el punto de vista del rendimiento. Los creadores de la tecnología se dieron cuenta de que la única vía correcta de implementar las consultas integradas contra un almacén relacional consiste en ir "capturando" en cada paso las especificaciones contenidas en cada una de las cláusulas `where`, `orderby`, etc., para solamente después analizada toda la cadena de llamadas a métodos extensores componer una única sentencia SQL a enviar al motor de base de datos.

Para este fin se definió una nueva interfaz, `IQueryable<T>`, que hereda de `IEnumerable<T>` (por lo cual los objetos que implementen esa interfaz son orígenes potenciales de consultas integradas), pero viene dotada de un conjunto de métodos extensores (cuya implementación básica está contenida en la clase estática `System.Linq.Queryable`, del mismo modo que los métodos extensores para `IEnumerable<T>` se implementan en `System.Linq.Enumerable`) que funcionan de un modo totalmente diferente a como lo hacen los operadores de LINQ to Objects, y que es más adecuada para el caso en que la fuente de datos a la que se desea acceder sea "remota", como ocurre en LINQ to SQL y LINQ to Entities.

La clase `Queryable` implementa prácticamente los mismos operadores de consulta que `Enumerable`, con una diferencia bastante lógica: en lugar de un operador `AsEnumerable()`, `Queryable` ofrece el operador `AsQueryable()`. Pero la diferencia más importante entre los métodos extensores de la clase `Enumerable` y los de `Queryable` estriba en que éstos últimos, en lugar de recibir como parámetros delegados (referencias de tipo `Func<T,...>`), reciben **árboles de expresiones** (referencias de tipo `Expression<Func<T,...>>`). Como conocerá el lector, las expresiones lambda pueden transformarse a conveniencia en unos u otros, por lo que una sentencia integrada como ésta:

```
var mayores20 = OrigenPersonas
                .Where(h => h.Edad > 20)
                .OrderBy(h => h.Nombre)
                .Select(h => new
                {
                    Nombre = h.Nombre.ToUpper(),
                    Edad = h.Edad
                });
```

compilará correctamente tanto si el origen de la consulta (`OrigenPersonas`) es `IQueryable<Persona>` como si es simplemente `IEnumerable<Persona>`. En el caso de las expresiones de consulta de LINQ to SQL, LINQ to Entities y otros proveedores basados en `IQueryable<T>`, las implementaciones de los operadores de consulta no reciben, conjuntamente con la secuencia de entrada, delegados a las funciones que hay que llamar según se estime oportuno, sino **árboles de expresiones que reflejan lo que estas funciones hacen**.

1.14.-¿Qué hacen los operadores de IQueryable<T>?

Para comprender cómo funcionan los proveedores basados en `IQueryable<T>`, hay que entender que todos los objetos que implementan esa interfaz llevan dentro de sí un arbol de expresión que refleja el algoritmo de obtención de la secuencia. En el caso de los objetos "iniciales" (los que sirven como origen a las consultas), estos contienen un árbol de un solo nodo, de tipo `ConstantExpression` (o sea, son "constantes").

Las implementaciones predeterminadas en `IQueryable<T>` de los operadores de consulta lo que hacen es generar un nuevo objeto `IQueryable<T>` que tiene como árbol de expresión el resultado de **combinar** el árbol de expresión del objeto de entrada con

los árboles de expresiones correspondientes a los demás argumentos del método. Por ejemplo, en el caso de la expresión de consulta anterior, el método `Where()` producirá un objeto de salida cuyo árbol de expresión tendrá como nodo raíz a un nodo de tipo `MethodCallExpression` (llamada a método), que a su vez tendrá, como primer argumento de la llamada, a la colección original, y como segundo, el árbol correspondiente al predicado a comprobar:

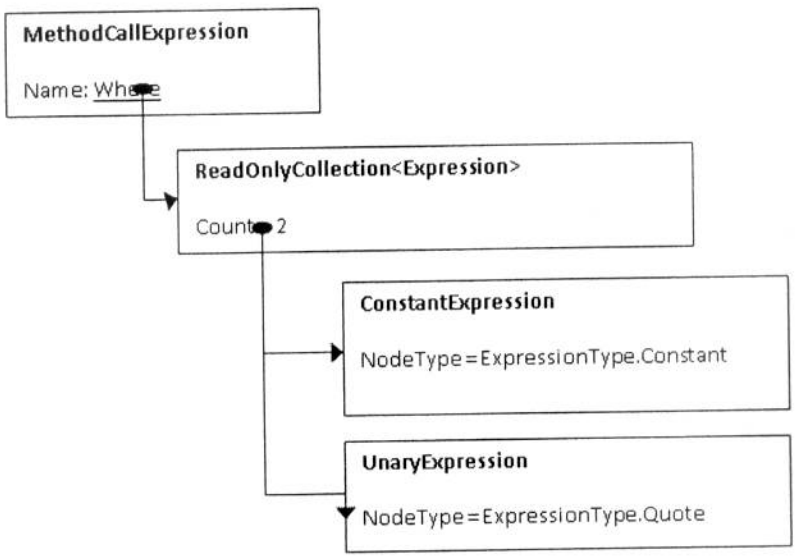


Figura A.2.- Árbol de un predicado

El mismo proceso se repetirá para los siguientes operadores de consulta (en el ejemplo, `OrderBy()` y `Select()`), sirviendo cada objeto `IQueryable<T>` resultante de una llamada anterior como objeto de entrada para la subsiguiente. Al final de la cadena, tendremos un árbol complejo que reflejará todo lo que la expresión de consulta debe realizar; éste árbol ya está listo para ser utilizado como fuente a partir de la cual construir, cuando se comience a iterar sobre los resultados de la consulta, una sentencia en el lenguaje del almacén remoto que se desea consultar (algún dialecto de SQL, en el caso de LINQ to SQL y LINQ to Entities). Para traducir los árboles de expresiones de LINQ en sentencias en el lenguaje del almacén, los proveedores LINQ se apoyan en clases auxiliares que se conocen como **proveedores de consultas** (*query providers*).

1.15.-Sobre la disponibilidad de operadores y funciones

Como se ha comentado anteriormente, en principio los únicos operadores de consulta estándar que deben ser implementados obligatoriamente por un proveedor de LINQ son el conjunto de sobrecargas de los operadores con reflejo en la sintaxis que en la especificación del lenguaje se conocen como patrón LINQ. Recuerde también que,

en el caso de los proveedores que hemos denominado "remotos", una expresión de consulta se traduce a fin de cuentas en un enorme árbol de expresiones, que posteriormente el proveedor de consultas correspondiente traducirá a una sentencia en el lenguaje del almacén remoto a consultar.

Para ciertos tipos de almacenes, algunos de los operadores de consulta estándar "extendidos" (de los que se invocan con notación funcional) pueden carecer de sentido o incluso ser imposibles de implementar. Por supuesto, en tales casos se deberá evitar el uso de dichos operadores; de hacerlo, se obtendrá una excepción de tipo `System.NotSupportedException`.

Como ejemplo, considere el caso del operador `ElementAt()`, que devuelve el elemento situado en una posición específica de la secuencia de entrada. Los creadores de LINQ to SQL y LINQ to Entities han entendido que no se debe permitir el uso de ese operador, por lo cual una consulta como la siguiente:

```
using (var ctx = new MAmazonEntities())
{
    // provoca una excepción
    var x = (from p in ctx.Producto
             orderby p.Titulo
             select p).ElementAt(2);
    Console.WriteLine(x.Titulo);
}
```

producirá una excepción. Observe, sin embargo, que la siguiente alternativa funciona correctamente:

```
using (var ctx = new MAmazonEntities())
{
    // funciona
    var y = (from p in ctx.Producto
             orderby p.Titulo
             select p).Skip(2).Take(1).First();
    Console.WriteLine(y.Titulo);
}
```

De manera similar, no se debe perder de vista el hecho de que la consulta finalmente deberá ser traducida en una sentencia del lenguaje del almacén contra el que se esté trabajando al escribir las expresiones lambda correspondientes a las cláusulas **`where`**, **`orderby`**, etc. de las consultas integradas. Por ejemplo, la siguiente consulta:

```
using (var ctx = new MAmazonEntities())
{
    // provoca una excepción
    var x = (from p in ctx.Producto
             // titulos de más de tres palabras
             where p.Titulo.
               Split(new char[] { ' ' }).Length > 3
             orderby p.Titulo
             select p.Titulo);
    foreach (var s in x)
        Console.WriteLine(s);
}
```

producirá una excepción, porque el proveedor de consultas de LINQ to Entities no tiene manera de expresar la llamada al método `Split()` de la clase `string` de .NET Framework en SQL, bastante más pobre que .NET en lo relativo al soporte de programación. Aquí se cumple el refrán "no se puede pedir peras al olmo" ☺.

Puede obtener toda la información relativa a los operadores y funciones soportados, soportados con ciertas limitaciones o totalmente no soportados por LINQ to SQL y LINQ to Entities en MSDN.

1.16.-Mecanismos de actualización

Para finalizar este apéndice, debemos mencionar, aunque sea "de pasada", que no obstante el hecho de que LINQ en principio solo tiene que ver con la recuperación de información, la mayoría de los proveedores LINQ han sido dotados de mecanismos adicionales para permitir también la actualización de datos y en general la manipulación, en el sentido más amplio de la palabra, de los contenedores en los que estos datos se almacenan, ya sean documentos XML, bases de datos relacionales u otros. En particular, las tecnologías basadas en LINQ que tienen que ver con el acceso a bases de datos relacionales (LINQ to SQL y LINQ to Entities) ofrecen la posibilidad de aplicar sobre el almacén relacional los cambios que se realicen a los objetos obtenidos a partir de la ejecución de consultas. Para ello, estas tecnologías son capaces de generar las sentencias INSERT, UPDATE y DELETE de SQL necesarias, como se muestra en el capítulo del libro en el que se describe LINQ to Entities.

APÉNDICE

Referencia de Entity SQL (eSQL)

1.- INTRODUCCIÓN

Entity SQL (**eSQL**) es un lenguaje de consulta similar a SQL y concebido especialmente para consultar modelos de datos de entidades (EDM) del Marco de entidades de ADO.NET (EF). En este anexo se muestran las principales características de eSQL y cómo construir sentencias de consulta, y se brinda una descripción de los principales operadores y funciones incorporados al lenguaje.

Para saber cómo ejecutar consultas eSQL desde código .NET, consulte los capítulos 3 y 4 de este libro.

Todos los ejemplos y los resultados de este apéndice se basan en una base de datos llamada **Mamazon** cuyos scripts de creación y de datos puede encontrar en la carpeta **sql** de los materiales que acompañan a este libro.

Nota: Exceptuando contadas ocasiones, que se mencionan expresamente, eSQL es insensible a las diferencias mayúsculas/minúsculas. Por consistencia, a lo largo de este apéndice, las palabras reservadas e identificador predefinido de eSQL se utilizan en mayúsculas.

1.1.- Diferencias con otros dialectos

Siguiendo la filosofía de EF, eSQL es agnóstico respecto al dialecto de SQL que utiliza el almacén de datos subyacente. El proveedor de EF para un cierto tipo de orígenes de datos (SQL Server, Oracle, DB2,Sqlite,etc.) es el encargado de transformar la consulta eSQL al dialecto SQL específico, por ejemplo, T-SQL para el caso de Sql Sever.

eSQL puede devolver objetos, colecciones y valores escalares, y por ello precisa de una semántica especial para poder tratar las colecciones y jerarquías de clases. Está

inspirado también en principios funcionales, y por ello admite cualquier expresión del lenguaje en lugares donde el SQL tradicional generalmente presenta limitaciones. En esta y en la anterior versión, carece de construcciones para la definición y manipulación de datos. Tampoco ofrece, por mostrar algunas lagunas, sentencias imperativas típicas de T-SQL, funciones analíticas, ni permite incluir pistas (*hints*) para la ejecución de consultas.

1.2.- Sentencias de consulta

Para construir sentencias de consulta en eSQL, se necesitan mecanismos para especificar la fuente para la consulta, las entidades o valores a devolver y los criterios a seguir para poder filtrarlos, agruparlos u ordenarlos. Para todo ello se utilizan unas cláusulas muy parecidas a las de SQL tradicional, que son, respectivamente, FROM, SELECT, WHERE, GROUP BY y ORDER BY.

Nota: En la documentación de MSDN para eSQL se utiliza normalmente el término "expresiones de consulta o *query expresions*" para referirse a las sentencias SELECT de eSQL. Aquí, hemos preferido utilizar el término "sentencias" para mantener la similitud con SQL y evitar confusiones.

1.2.1.- FROM

La cláusula `FROM` permite especificar una expresión que servirá como fuente de datos en la sentencia `SELECT`. Los nombres o alias definidos en la cláusula `FROM` son procesados de izquierda a derecha. Si se utiliza la cláusula `JOIN`, se puede hacer referencia a los alias definidos anteriormente. Las expresiones construidas aquí no pueden (al contrario que en SQL) utilizarse directamente en el resto de cláusulas, en las que debe emplearse obligatoriamente su alias. Por ejemplo, las siguientes serían consultas eSQL sintácticamente **incorrectas**:

```
select Producto from MamazonEntities.Producto;
select * from MamazonEntities.Producto;
from MamazonEntities.Producto
```

La sentencia correcta para obtener todos los productos debe definir obligatoriamente un alias:

```
select p from MamazonEntities.Producto as p;
```

Aunque esta expresión de consulta también sería correcta, porque utiliza la generación automática de alias:

```
MamazonEntities.Producto;
```

FROM soporta los encuentros internos (INNER JOIN), encuentros externos por la izquierda y la derecha (LEFT y RIGHT OUTER JOIN) y encuentros externos simétricos (FULL OUTER JOIN), además de los encuentros cruzados o productos cartesianos (CROSS JOIN). También soporta CROSS APPLY y OUTER APPLY, pero el uso de éstos solo es posible cuando el almacén de datos sea SQL Server 2005 o superior.

La siguiente consulta devuelve el producto cartesiano de los productos y distribuidores

```
select producto, distribuidor
  from MamazonEntities.Producto as producto cross join
MamazonEntities.Entidad as distribuidor
```

Al contrario que otros dialectos SQL en Entity Framework las relaciones son entidades de primer orden, y si el EDM está correctamente definido es posible navegar cómodamente por ellas sin necesidad de hacer *joins*.

1.2.2.- SELECT

La cláusula **SELECT** permite indicar los valores que se devuelven de una consulta. Es evaluada después de que las cláusulas **FROM**, **GROUP BY** y **HAVING** hayan sido evaluadas. Normalmente las expresiones presentes en la cláusula **SELECT** se construyen utilizando los alias definidos en la cláusula **FROM**, pero si se ha utilizado un **GROUP BY** en la consulta, solo se podrá hacer referencia a las claves de grupo (*keys*) definidas en dicha cláusula o incluir expresiones basadas en funciones de agregación.

Si la consulta devuelve valores duplicados, al igual que en SQL puede utilizarse el modificador **DISTINCT**, que los elimina:

```
select distinct producto.GeneroPrincipal from
MamazonEntities.Producto as producto
```

GeneroPrincipal
Acción
Bélica
Clásica

Nota: Los resultados de las consultas de ejemplo se mostrarán en tablas a continuación de cada consulta. Algunos resultados son complicados de representar en un libro, por lo que algunas tablas pueden ser difíciles de visualizar correctamente o se verán los datos bastante comprimidos. Le pedimos disculpas de antemano.

El modificador `TOP` permite devolver solo el número de ocurrencias especificadas. A menos que se utilice `ORDER BY` los resultados que devuelve `TOP` no son deterministas. Si lo que se desea es paginar los resultados es recomendable utilizar los modificadores `SKIP` y `LIMIT` de `ORDER BY` (ver `ORDER BY`).

Por ejemplo:

```
select top(2) producto from MamazonEntities.Producto as producto
```

Producto							
Id Producto	**Titulo**	**Genero Principal**	**Porcentaje Distribuidor**	**Descripcion**	**Imagen**	**Fecha Lanza-miento**	**Sub Tipo**
1000	Solo	Indie	1,00	NULL	NULL	02/04/2001 0:00:00	A
Id Producto	**Titulo**	**Genero Principal**	**Porcentaje Distribuidor**	**Descripcion**	**Imagen**	**Fecha Lanza-miento**	**Sub Tipo**
1001	Coches sin ruedas	Indie	2,00	NULL	NULL	02/04/2001 0:00:00	A

Recordemos que, a diferencia de los dialectos SQL, la cláusula `SELECT` en ESQL no soporta el "*" como operador de selección de atributos, en ESQL se necesita referenciar obligatoriamente al alias definido en el FROM o crear una expresión.

Además, en ESQL hay dos variantes de la cláusula `SELECT` a la hora de proyectar los valores que devuelve; `SELECT ROW(...)` y `SELECT VALUE`. La primera variante es implícita (puede evitarse utilizar ROW()) y devuelve un conjunto múltiple de valores. Si no se especifica un alias para cada valor, se genera un alias automático. Las consultas en un dialecto SQL normal no devuelven conjuntos múltiples de datos. Para que el `SELECT` se comporte de forma parecida a SQL en ESQL debe utilizarse la cláusula `SELECT VALUE` que devuelve un único valor en forma de fila sin valores múltiples y sus atributos tienen un alias con su nombre en el espacio C.

Veamos varios ejemplos:

```
select value producto from MamazonEntities.Producto as producto
```

Id Producto	**Titulo**	**Genero Principal**	**Porcentaje Distribuidor**	**Descripcion**	**Imagen**	**Fecha Lanza-miento**	**Sub Tipo**
1000	Solo	Indie	1,00	NULL	NULL	02/04/2001 0:00:00	A
1001	Coches sin ruedas	Indie	2,00	NULL	NULL	02/04/2001 0:00:00	A

```
select producto from MamazonEntities.Producto as producto
```

Producto

Id Producto	Titulo	Genero Principal	Porcentaje Distribuidor	Descripcion	Imagen	Fecha Lanzamiento	Sub Tipo
1000	Solo	Indie	1,00	NULL	NULL	02/04/2001 0:00:00	A
Id Producto	**Titulo**	**Genero Principal**	**Porcentaje Distribuidor**	**Descripcion**	**Imagen**	**Fecha Lanzamiento**	**Sub Tipo**
1001	Coches sin ruedas	Indie	2,00	NULL	NULL	02/04/2001 0:00:00	A

Se observa como la segunda consulta, que lleva un recubrimiento implícito de ROW, devuelve un conjunto múltiple de resultados.

En la siguiente consulta se observa la potencia de ESQL para devolver conjunto de valores:

```
select producto, producto.Distribuidor from MamazonEntities.Producto
as producto
```

Producto

Id Producto	Titulo	Genero Principal	Porcentaje Distribuidor	Descripcion	Imagen	Fecha Lanzamiento	Sub Tipo
1000	Solo	Indie	1,00	NULL	NULL	02/04/2001 0:00:00	A
Id Producto	**Titulo**	**Genero Principal**	**Porcentaje Distribuidor**	**Descripcion**	**Imagen**	**Fecha Lanzamiento**	**Sub Tipo**
1001	Coches sin ruedas	Indie	2,00	NULL	NULL	02/04/2001 0:00:00	A

Distribuidor

idEntidad	NombreGrupo	Nombre	Apellidos	EsUnGrupo	SubTipo	PorcentajeDefecto	IdentificadorFiscal
300	Krasis Press	NULL	NULL	True	D	10,25	X21342 3423B
idEntidad	**NombreGrupo**	**Nombre**	**Apellidos**	**EsUnGrupo**	**SubTipo**	**PorcentajeDefecto**	**IdentificadorFiscal**
300	Krasis Press	NULL	NULL	True	D	10,25	X21342 3423B

Si en la consulta anterior utilizamos SELECT VALUE obtendríamos una excepción al no soportar valores múltiples.

WHERE

La cláusula WHERE se procesa justo a continuación de la cláusula FROM y espera una expresión lógica. Recordemos que las expresiones pueden componerse con distintos operadores y funciones que se pueden consultar en los apartados correspondientes. La

expresión WHERE filtra o limita los valores referenciados en la cláusula FROM, por ejemplo la siguiente consulta:

```
SELECT TOP(2) producto
FROM MamazonEntities.Producto AS producto
WHERE producto.idProducto = 1000
```

Devolverá:

Producto							
Id Producto	Titulo	Genero Principal	Porcentaje Distribuidor	Descripcion	Imagen	Fecha Lanza-miento	Sub Tipo
1000	Solo	Indie	1,00	NULL	NULL	02/04/2001 0:00:00	A

1.2.3.- GROUP BY

Sirve para agrupar valores de la cláusula SELECT y así poder utilizar funciones de agrupamiento como COUNT, SUM, etc. En cuanto se usa una cláusula GROUP BY las cláusulas SELECT y HAVING solo pueden referenciar a los alias definidos en GROUP BY y los alias del FROM ya no están accesibles excepto para las funciones de agrupamiento.

Para filtrar los valores que devuelve una cláusula GROUP BY se utiliza la cláusula HAVING

Por ejemplo, la siguiente consulta obtendría los tres géneros con más productos y su número:

```
select top(3) grpGenero as Generos, count(producto.GeneroPrincipal)
as cantidad
    from MamazonEntities.Producto as producto
    group by producto.GeneroPrincipal as grpGenero
    order by cantidad desc
```

Géneros	Cantidad
Rock	7
Comedia	6
Easy Listening	6

La cláusula SELECT utiliza el alias definido en el GROUP BY, pero la función de agrupamiento puede utilizar el alias definido en el FROM.

1.2.4.- HAVING

Especifica una condición de búsqueda para un grupo o un agregado en aquellas consultas ESQL que contienen una cláusula GROUP BY. Funciona igual que el WHERE pero se aplica después de procesar GROUP BY.

Por ejemplo, la siguiente consulta:

```
select top(3) grpGenero as Generos, count(producto.GeneroPrincipal)
as cantidad
    from MamazonEntities.Producto as producto
   group by producto.GeneroPrincipal as grpGenero
    having count(producto.GeneroPrincipal) > 6
   order by cantidad desc
```

devolvería una única tupla:

Géneros	Cantidad
Rock	7

1.2.5.- ORDER BY

La cláusula opcional `ORDER BY` especifica cuál es el orden en el que se deben devolver los valores que proyectará `SELECT`. Además puede especificarse un número de ocurrencias que deben saltarse (`SKIP`) y limitar los resultados devueltos (`LIMIT`) con lo que es muy fácil implementar una paginación efectiva. Como en SQL se puede especificar varias expresiones de orden y si debe ordenarse de forma ascendente o descendente con `ASC` y `DESC` (por defecto ascendente). Para referenciar el orden se utilizan expresiones construidas, por ejemplo, con los alias definidos en la cláusula `SELECT`.

Veamos varios ejemplos de consultas con ordenación:

```
select persona from MamazonEntities.Entidad as persona
order by persona.Apellidos + persona.Nombre;

select producto.IdProducto, producto.Titulo from
MamazonEntities.Producto as producto order by
producto.PorcentajeDistribuidor desc skip 0 limit 10;
```

1.2.6.- Expresiones

En esta sección se describen las posibilidades que ofrece eSQL para la creación de expresiones para las cláusulas `WHERE`, `GROUP BY`, etc. de las sentencias de consulta.

Parámetros

Es posible (y generalmente muy útil) parametrizar las consultas eSQL. Esto se hace de forma muy parecida a Transact SQL, y tiene sentido siempre que tengamos una API de nivel superior que invoque la consulta (por ejemplo, Entity Client). Los parámetros se especifican utilizando el carácter arroba (`@`), y pueden utilizarse en cualquier punto

de la sentencia donde se acepte una expresión, como en el caso de las cláusulas `WHERE`, `TOP`, `LIMIT` o `SKIP`. El siguiente es un ejemplo de sentencia parametrizada:

```
SELECT producto.IdProducto, producto.Titulo
FROM MamazonEntities.Producto AS producto
WHERE producto.Titulo LIKE @like_param
ORDER BY producto.PorcentajeDistribuidor DESC
  SKIP @skip_param LIMIT @limit_param;
```

Operadores

El lenguaje eSQL proporciona un conjunto de operadores para construir expresiones. Algunos operadores son similares a los de SQL, pero otros están orientados a navegar por las jerarquías y las relaciones entre entidades. Se admite el uso de expresiones en prácticamente todas las cláusulas de las sentencias de eSQL.

Operadores aritméticos

eSQL dispone de los operadores aritméticos más comunes:

Operador	Descripción
+	Adición, aplicado a dos valores numéricos. Concatenación, aplicado a cadenas de caracteres.
-	Sustracción, aplicado a dos valores numéricos. Negación, aplicado a un único valor numérico.
*	Multiplicación, aplicado a dos valores numéricos.
/	División, aplicado a dos valores numéricos.
%	Módulo o resto de la división entera. Aplicable únicamente a dos valores numéricos enteros.

Un ejemplo de consulta con expresiones numéricas sería:

```
SELECT distribuidor.NombreGrupo, distribuidor.PorcentajeDefecto,
  precio AS PrecioProducto,
  precio*distribuidor.PorcentajeDefecto/100 AS GananciaDistribuidor,
  precio*(1-distribuidor.PorcentajeDefecto/100) AS GananciaFabricante
FROM OFTYPE(MamazonEntities.Entidad, MamazonModel.Distribuidor)
  AS distribuidor, {1500} AS precio
ORDER BY GananciaDistribuidor DESC
```

El resultado es:

NombreGrupo	PorcentajeDefecto	PrecioProducto	GananciaDistribuidor	GananciaFabricante
Trono Productions	73,00	1500	1095,000000	405,000000
Maikel Naig Software	33,33	1500	499,950000	1000,050000
Centroña Producciones	23,00	1500	345,000000	1155,000000
Indie Group	20,00	1500	300,000000	1200,000000
Carolco Pictures	15,00	1500	225,000000	1275,000000

PlainConcepts Software Factory	12,05	1500	180,750000	1319,250000
MEMI España	11,10	1500	166,500000	1333,500000
Krasis Press	10,25	1500	153,750000	1346,250000
Nilo Books	5,80	1500	87,000000	1413,000000
SONI VGM	5,23	1500	78,450000	1421,550000
Happy Music	5,10	1500	76,500000	1423,500000
Pia Alternative Music	4,20	1500	63,000000	1437,000000
Guarner Ltd	2,30	1500	34,500000	1465,500000
Ileeteea Editors	2,00	1500	30,000000	1470,000000
Memola Films	0,90	1500	13,500000	1486,500000
Martin Flowers Records	0,10	1500	1,500000	1498,500000

La consulta, además de operadores aritméticos, utiliza el operador de tipo `OFTYPE`, que se describe más adelante.

Operadores de comparación

Los operadores de comparación en eSQL son muy parecidos a los de SQL; se aplican sobre expresiones y tienen un resultado lógico. Se suelen utilizar en la cláusula `WHERE`.

Operador	Descripción
=	Comprueba la igualdad de dos expresiones.
>	Comprueba si la primera expresión es mayor que la segunda.
>=	Comprueba si la primera expresión es mayor o igual que la segunda.
IS [NOT] NULL	Comprueba si una expresión es (no es) nula.
<	Comprueba si la primera expresión es menor que la segunda.
<=	Comprueba si la primera expresión es mayor que la segunda.
[NOT] BETWEEN	Comprueba si una expresión está (no está) en un rango de valores.
!= ó <>	Comprueba si dos expresiones no son iguales.
[NOT] LIKE	Determina si una expresión cumple (no cumple) un patrón LIKE de SQL específico.

La siguiente consulta devuelve todos los proveedores que contienen la letra 'o' en su nombre y tienen un porcentaje mayor del 10%.

```
SELECT distribuidor.NombreGrupo, distribuidor.PorcentajeDefecto
FROM OFTYPE(MamazonEntities.Entidad, MamazonModel.Distribuidor)
  AS distribuidor
WHERE distribuidor.NombreGrupo LIKE '%o%'
  AND distribuidor.porcentajeDefecto > 10
```

El resultado es:

NombreGrupo	PorcentajeDefecto
PlainConcepts Software Factory	12,05
Indie Group	20,00
Carolco Pictures	15,00
Trono Productions	73,00
Centroña Producciones	23,00
Maikel Naig Software	33,33

Operadores lógicos

Los operadores lógicos de eSQL son los tradicionales de un dialecto SQL: AND (conjunción, que también puede escribirse al "estilo C#", mediante &&), OR (disyunción, también representable mediante ||) y NOT (negación, cuyo sinónimo es !).

Operadores de acceso a miembros

Existe un único operador de acceso a los miembros de una entidad del modelo: el operador punto (.).

Operadores de tipo

Los operadores de tipo son especialmente importantes en eSQL. A excepción de CAST, que le resultará más familiar, los demás operadores de tipo (IS [NOT] OF, OFTYPE y TREAT) operan directamente sobre un modelo EDM y la jerarquía de clases que éste contiene.

★ CAST

El operador CAST intenta convertir una expresión a un tipo dado, de manera similar al operador CONVERT de Transact SQL. Su sintaxis es:

```
CAST(<expresión> AS <tipo>)
```

Si no es posible realizar la conversión, se producirá una excepción. El tipo debe ser un tipo primitivo (escalar) o una enumeración. Si la consulta se ejecuta a través de un `EntityCommand`, el tipo debe pertenecer al espacio de nombres de EDM (`Edm.Int32`, `Edm.Decimal`, etc.). Si la consulta se ejecuta a través de `ObjectQuery<T>`, el tipo puede también ser un tipo común del CLR.

En el siguiente ejemplo se obtienen los porcentajes por defecto de los distribuidores, convirtiendo el tipo numérico a un decimal de 4 dígitos y 2 decimales.

```
SELECT distribuidor.NombreGrupo,
  CAST(distribuidor.PorcentajeDefecto AS  Edm.Decimal(4,2))
    AS porcentaje
FROM OFTYPE(MamazonEntities.Entidad, MamazonModel.Distribuidor)
  AS distribuidor
WHERE distribuidor.porcentajeDefecto > 20
ORDER BY porcentaje DESC
```

El resultado es:

NombreGrupo	Porcentaje
Trono Productions	73,00
Maikel Naig Software	33,33
Centroña Producciones	23,00

★ IS [NOT] OF y TREAT

El operador `IS OF` comprueba si una expresión es de un determinado tipo o subtipo. La sintaxis exacta es:

```
<expresión> IS [NOT] OF ([ONLY] <tipo>)
```

El modificador `ONLY` indica que la expresión debe ser exactamente del tipo especificado, y no de ninguno de sus subtipos. El modificador `NOT` invierte el sentido de la comprobación de tipo.

Por su parte, el operador `TREAT` permite tratar un objeto de un tipo base específico como de un tipo derivado de él, de modo similar a como funcionan el operador `as` de C#. Su sintaxis es:

```
TREAT(<expresión> AS <tipo>)
```

Ambos operadores se suelen utilizar conjuntamente: con `IS OF` se hace una comprobación de tipo en la cláusula `WHERE`, y con `TREAT` se produce ese subtipo en la cláusula `SELECT`. Por ejemplo:

```
SELECT VALUE TREAT(persona AS MamazonModel.Distribuidor)
  FROM MamazonEntities.Entidad AS persona
 WHERE persona IS OF (MamazonModel.Distribuidor);
```

idEntidad	NombreGrupo	Nombre	Apellidos	EsUnGrupo	SubTipo	PorcentajeDefecto	IdentificadorFiscal
300	Krasis Press	**NULL**	**NULL**	True	D	10,25	X2134234238
301	MEMI España	**NULL**	**NULL**	True	D	11,10	S2323424340

Esta consulta devolverá solo los distribuidores con los atributos propios de `Distribuidor` y de `Entidad`, del que aquél es subtipo.

★ OFTYPE

El operador `OFTYPE` se aplica sobre una expresión de consulta y devuelve una colección con los elementos que son del tipo específico. Su sintaxis es:

```
OFTYPE(<expresión>, [ONLY] <tipo>)
```

El uso de `OFTYPE` equivale a de la combinación de `IS OF` y `TREAT` presentada anteriormente:

```
SELECT value distribuidor
FROM OFTYPE(MamazonEntities.Entidad, MamazonModel.Distribuidor)
  AS distribuidor;
```

Operadores de conjuntos

eSQL proporciona un grupo de operadores de conjuntos sumamente útiles: `UNION`, `INTERSECT`, `EXCEPT` y `EXISTS`. También tiene operadores para eliminar duplicados (`SET`), comprobar pertenencia (`IN`), entre otros. La siguiente tabla describe estos operadores.

Operador	Descripción
ANYELEMENT	Recupera un único valor de un conjunto. Sintaxis: `ANYELEMENT(<colección>)` Ejemplo: `ANYELEMENT(SELECT value persona FROM MamazonEntities.Entidad AS persona)`
EXCEPT	Operación de diferencia de conjuntos: elimina del primer conjunto los valores que se encuentren en el segundo. Sintaxis: `<colección> EXCEPT(<colección>)` Ejemplo: `(SELECT producto.Titulo, producto.GeneroPrincipal FROM MamazonEntities.Producto AS producto WHERE producto.PorcentajeDistribuidor > 2.2m) EXCEPT (SELECT producto.Titulo, producto.GeneroPrincipal FROM MamazonEntities.Producto AS producto WHERE producto.porcentajeDistribuidor > 3.0m)`
FLATTEN	"Aplana" una colección que puede contener otras colecciones anidadas en una colección de un solo nivel de profundidad, sin anidamientos. Sintaxis: `FLATTEN(<colección>)` Ejemplo: Esta consulta devuelve para cada distribuidor todos los productos que distribuye: `SELECT VALUE distribuidor.Producto FROM OFTYPE(MamazonEntities.Entidad, MamazonModel.Distribuidor) AS distribuidor`

	Utilizando **FLATTEN**, se obtendrá una única colección con todos los productos distribuidos por los distribuidores: `FLATTEN (SELECT VALUE distribuidor.Producto FROM OFTYPE(MamazonEntities.Entidad, MamazonModel.Distribuidor) AS distribuidor)`
INTERSECT	Operación de intersección de dos conjuntos: produce los elementos comunes a los dos conjuntos. Sintaxis: `<colección> INTERSECT(<colección>)`
[NOT] EXISTS	Devuelve verdadero si la colección contiene elementos, y falso en caso contrario. El modificador **NOT** niega el resultado de la comparación. Sintaxis: `[NOT] EXISTS(<colección>)`
[NOT] IN	Devuelve verdadero si el valor de la izquierda pertenece a la colección de la derecha, y falso en caso contrario. El modificador **NOT** niega el resultado de la comparación. Sintaxis: `<valor> [NOT] IN (<colección>)`
OVERLAPS	Determina si la primera colección tiene elementos en común con la segunda. Sintaxis: `<colección> OVERLAPS(<colección>)` Equivale a: `EXISTS(<colección> INTERSECT <colección>)`
SET	Convierte una colección de objetos en un conjunto, eliminando los valores duplicados que pudiera contener. Sintaxis: `SET(<colección>)` Equivale a: `SELECT VALUE DISTINCT <expresión> FROM <expression>`
UNION	Combina (une) dos colecciones en una. Los valores duplicados se suprimen, a menos que se especifique la cláusula **ALL**. Sintaxis: `<colección> UNION(<colección>)`

Operadores de construcción de tipos

eSQL proporciona tres constructores (mecanismos de construcción) de tipo: los **constructores de fila**, los **constructores de tipos por nombre** y los **constructores de colecciones**.

a) Para la construcción de filas se utiliza la palabra reservada **ROW**. Los constructores de filas permiten crear registros estructurados de un tipo anónimo a partir de uno o más valores. Por ejemplo, la siguiente expresión produce un valor del tipo `Record(a int, b string, c int)`:

```
ROW(1 AS a, "mamazon" AS b, 16 AS c);
```

b) Para la construcción de tipos por nombre, se utiliza el nombre del tipo del EDM. Pueden crearse instancias de tipos complejos y de tipos de entidad. Los argumentos

para la construcción pueden ser expresiones y deben suministrarse en el mismo orden en que las propiedades están declaradas en la entidad. Por ejemplo:

```
MamazonModel.Distribuidor(1, null,
  "Manuel", "Barreto", false, "E", 0.2m,
  "32433423M");
```

Esta expresión crea una instancia del tipo `MamazonModel.Distribuidor`. Los primeros argumentos coinciden con los de `MamazonModel.Entidad`, la clase base de `Distribuidor` en el modelo EDM, y los dos últimos valores se asociarán a `PorcentajeDefecto` e `IdentificaciónFiscal`, las propiedades propias del distribuidor.

c) La construcción de colecciones se puede hacer mediante el operador **MULTISET**, o simplemente declarando entre llaves el conjunto de valores. Por jemplo, las dos siguientes expresiones son equivalentes:

```
MULTISET(1,2,3)
{1, 2, 3}
```

Operadores de referencia

Una referencia es un puntero lógico (análogo a una clave foránea en un modelo relacional) a una entidad específica de un conjunto de entidades. En la práctica, es una estructura que contiene los atributos y valores mínimos que identifican unívocamente a la entidad, y suele coincidir con la clave primaria del almacén subyacente. eSQL ofrece una serie de operadores para manipular referencias: **CREATEREF**, **DEREF**, **KEY** y **REF**.

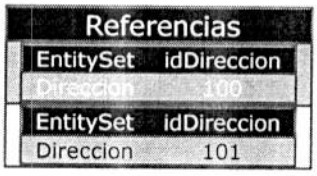
★ CREATEREF

El operador **CREATEREF** fabrica una referencia a partir de un identificador de conjunto de entidades y una fila con valores tipados correspondientes a los valores de las propiedades de identidad de la fila. La sintaxis es:

```
CREATEREF(<EntitySet>, <constructor_fila>)
```

Ejemplo:

```
SELECT CREATEREF(MamazonEntities.Direccion, ROW(e.IdEntidad))
FROM MamazonEntities.Entidad AS e
```



Esta consulta crearía un conjunto de referencias de dirección a partir de los valores de `idEntidad` de la entidad `Entidad` (que representa en el EDM de ejemplo a personas). Más adelante, se podrá utilizar el operador **DEREF** para obtener la entidad `Direccion` correspondiente a dicha referencia. Es importante entender que esa `Direccion` podría no existir; en tal caso, **DEREF** devolvería el valor nulo.

En el siguiente ejemplo, creamos una referencia a una entidad `Entidad` con identificador 300, y accedemos su atributo `NombreGrupo`:

```
CREATEREF(MamazonEntities.Entidad, ROW(300)).NombreGrupo
```

En este momento, al lector le puede surgir la duda sobre qué es más aconsejable a la hora de recuperar una entidad por su identificador: si utilizar **CREATEREF** o simplemente ejecutar la siguiente consulta equivalente:

```
SELECT e.NombreGrupo
FROM MamazonEntities.Entidad AS e
WHERE e.IdEntidad=300
```

Ante todo, esta segunda forma es más entendible y elegante; pero además, si se analiza el código Transact SQL que se envía a un motor de SQL Server como resultado de la ejecución de ambas consultas, se verá con claridad que la segunda variante es preferible desde el punto de vista del rendimiento.

★ REF

El operador **REF** obtiene una referencia a una entidad. Una referencia consiste en la clave de la entidad y el nombre del conjunto al que pertenece. La sintaxis de este operador es:

```
REF(<expresión>)
```

Si la expresión de entrada representa a una entidad que está guardada en el almacén de datos, el operador devolverá una referencia a esta entidad. En caso contrario, se devolverá una referencia nula. Por ejemplo, la consulta:

```
SELECT REF(p) AS RefProducto FROM MamazonEntities.Producto AS p
```

produce como resultado:

RefProducto	
EntitySet	idProducto
Producto	1144
EntitySet	idProducto
Producto	1101

Al igual que en el caso de **CREATEREF**, si se utiliza el operador de acceso a miembro (.) para acceder a una propiedad de la entidad, la referencia se resuelve automáticamente.

★ DEREF

El operador **DEREF** resuelve una referencia; o sea, obtiene la entidad a la que la referencia "apunta". La sintaxis de este operador es:

```
DEREF(<expresión>)
```

Si la referencia es nula, o no existe la entidad referenciada, DEREF devuelve el valor nulo.

Como ejemplo, la consulta:

```
SELECT DEREF(REF(p)) AS RefProducto
FROM MamazonEntities.Producto AS p
```

es equivalente a:

```
SELECT p AS RefProducto
FROM MamazonEntities.Producto AS p
```

★ KEY

El operador **KEY** extrae el valor de la clave de identidad a partir de una referencia o de una expresión que devuelve una entidad. Su sintaxis es:

```
KEY(<expresión>)
```

El tipo de retorno es un tipo de fila con un campo por cada una de las propiedades que componen la clave de la entidad.

Por ejemplo, si ejecutamos:

```
SELECT KEY(p) AS key_p,
  KEY(REF(p)) AS key_ref_p,
  KEY(CREATEREF (MamazonEntities.Producto, ROW(p.IdProducto)))
    AS key_createref
FROM MamazonEntities.Producto AS p
```

obtendríamos:

key_p	key_ref_p	key_createref
idProducto: 1144	idProducto: 1144	idProducto: 1144
idProducto: 1101	idProducto: 1101	idProducto: 1101

Operador de navegación

El operador NAVIGATE permite navegar por las relaciones entre entidades. En eSQL, las relaciones entre objetos son entidades de primer orden. La sintaxis de este operador es:

```
NAVIGATE(<expresión de instancia>, [<tipo relación>],
  [<hacia> [, <desde>]])
```

La <expresión de instancia> es un alias proveniente de la cláusula FROM de la consulta. El <tipo relación> es una relación definida en el modelo de entidades. También se pueden especificar el destino y el origen de la relación. Si la cardinalidad del destino de la relación (representado por <hacia>) es 1, devolverá una referencia del tipo de destino. En caso de que esta cardinalidad sea N, se devolverá una colección de referencias.

Por ejemplo, la siguiente consulta eSQL devuelve el código y nombre de un Producto junto a una referencia a su Distribuidor, que se obtiene a través del operador de navegación.

```
SELECT p.IdProducto, p.Titulo,
  NAVIGATE(p, MamazonModel.DistribuidorProducto) AS DistribuidorRef
FROM MamazonEntities.Producto AS p
```

El resultado sera:

IdProducto	Titulo	DistribuidorRef	
1000	Solo	EntitySet	idEntidad
		Entidad	300
1001	Coches sin ruedas	EntitySet	idEntidad
		Entidad	300
1002	Lack of lucky	EntitySet	idEntidad
		Entidad	300

Obviamente, podemos utilizar los operadores de referencia comentados en el apartado anterior para recuperar entidades completas:

```
SELECT p.IdProducto, p.Titulo,
  DEREF(NAVIGATE (p, MamazonModel.DistribuidorProducto))
  AS Distribuidor
FROM MamazonEntities.Producto AS p
```

En este caso, el resultado será:

IdProducto	Titulo	Distribuidor							
		idEntidad	NombreGrupo	Nombre	Apellidos	EsUnGrupo	SubTipo	PorcentajeDefecto	IdentificadorFiscal
1000	Solo	300	Krasis Press	NULL	NULL	True	D	10,25	X213423423B
1001	Coches sin ruedas	300	Krasis Press	NULL	NULL	True	D	10,25	X213423423B
1002	Lack of lucky	300	Krasis Press	NULL	NULL	True	D	10,25	X213423423B

Funciones canónicas

eSQL define una serie de **funciones canónicas**, que en principio son válidas para cualquier origen de datos y que deben ser implementadas por cada proveedor de EF concreto, que las transformará en llamadas a las funciones específicas del lenguaje de consultas del almacén de datos. No se puede garantizar, sin embargo, que todos los proveedores implementen todas las funciones canónicas de eSQL.

Las funciones pueden clasificarse en funciones de agregación, matemáticas, de manipulación de cadenas, de manipulación de fecha y hora, de manipulación de bits y otras. A continuación se describen someramente estas funciones.

Funciones de agregación

Las funciones de agregación son expresiones que reducen un conjunto de elementos de entrada a un único valor "acumulado". Se utilizan frecuentemente en combinación con la cláusula `GROUP BY`.

Función	Descripción
AVG	Devuelve la media aritmética de los valores no nulos. Sintaxis: `AVG(<expresión>)` Ejemplo: La siguiente consulta devuelve la duración media de los productos de audio agrupados por género. Observe el uso de variables en las expresiones y su uso posterior en las cláusulas `SELECT`, `GROUP BY` y `ORDER BY`, que no son posibles en Transact SQL y la mayoría de otros dialectos.

```
SELECT generoDelProducto AS Genero,
  ROUND(AVG(
    TREAT(producto AS Model.Audio).Duracion)/60)
  AS DuracionMedia
FROM MamazonEntities.Producto AS producto
WHERE producto IS OF (Model.Audio)
GROUP BY producto.GeneroPrincipal
  AS generoDelProducto
ORDER BY DuracionMedia desc
```

Genero	DuracionMedia
Diversion	14
Deportes	12
Noticias	12
Easy Listening	8
Rumba	6
Indie	5
Pop	5
Rap	4
Copla	4
Rock	2
Ole	2

BIGCOUNT Devuelve la cantidad de elementos que componen una expresión, incluyendo valores nulos y duplicados (el tipo devuelto es `Int64`).
Sintaxis: `BIGCOUNT(<expresión>)`

COUNT Devuelve la cantidad de elementos que componen una expresión, incluyendo valores nulos y duplicados (el tipo devuelto es `Int32`).
Sintaxis: `COUNT(<expresión>)`
Ejemplo: La siguiente consulta devuelve el número de productos de vídeo.

```
SELECT count(producto.IdProducto)
FROM MamazonEntities.Producto AS producto
WHERE producto IS OF (Model.Video)
```

10

MAX Devuelve el valor máximo de los valores no nulos.que componen la expresión. Además de tipos numéricos, permite fechas y y cadenas de caracteres.
Sintaxis: `MAX(<expresión>)`
Ejemplo: La siguiente consulta devuelve la fecha del último lanzamiento de una película o serie.

```
SELECT MAX(
  TREAT(producto AS Model.Video).FechaLanzamiento)
FROM MamazonEntities.Producto AS producto
WHERE producto is of (Model.Video)
```

07/10/2008 0:00:00

MIN	Devuelve el valor mínimo de los valores no nulos.que componen la expresión. Además de tipos numéricos, permite fechas y y cadenas de caracteres. Sintaxis: `MIN(<expresión>)`
STDEV	Devuelve la desviación típica de los valores no nulos.que componen la expresión. Sintaxis: `STDEV(<expresión>)`

SUM	Devuelve la suma de los valores no nulos.que componen la expresión.
	Sintaxis: `SUM(<expresión>)`
	Ejemplo: La siguiente consulta devuelve la duración total de todas las películas o series.
	`SELECT SUM(`
	`  TREAT(producto AS Model.Video).Duracion)`
	`FROM MamazonEntities.Producto AS producto`
	`WHERE producto is of (Model.Video)`
	
	1088

Antes de seguir describiendo el resto de funciones canónicas, queremos volver a incidir sobre el hecho de que eSQL está permeado por influencias de la programación funcional, y a diferencia de los dialectos tradicionales de SQL acepta expresiones más complejas, por lo que son posibles expresiones del tipo:

```
MAX( SELECT value treat(producto AS Model.Video).FechaLanzamiento
     FROM MamazonEntities.Producto AS producto
     WHERE producto is of (Model.Video) );
```

Que en el caso de SQL Server genera la siguiente sentencia de Transact SQL:

```
SELECT [GroupBy1].[A1] AS [C1]
FROM (
  SELECT cast(1 AS bit) AS X ) AS [SingleRowTable1]
  LEFT OUTER JOIN (
    SELECT MAX([Extent1].[FechaLanzamiento]) AS [A1]
    FROM  [dbo].[Producto] AS [Extent1]
    INNER JOIN [dbo].[Video] AS [Extent2]
    ON [Extent1].[idProducto] = [Extent2].[idProducto] )
  AS [GroupBy1]
  ON 1 = 1
```

Si se compara la sentencia anterior con la sentencia Transact SQL generada para la consulta "tradicional" (con la llamada a la función `MAX` dentro de la cláusula `SELECT`):

```
SELECT [GroupBy1].[A1] AS [C1]
FROM (
```

```
    SELECT MAX([Extent1].[FechaLanzamiento]) AS [A1]
    FROM  [dbo].[Producto] AS [Extent1]
    INNER JOIN [dbo].[Video] AS [Extent2]
    ON [Extent1].[idProducto] = [Extent2].[idProducto] )
AS [GroupBy1]
```

Se verá que en el primer caso se genera una sentencia SELECT adicional con respecto a la segunda consulta, si bien su cómputo es despreciable.

Los creadores de EF recomiendan utilizar las funciones de agregación sobre colecciones preferentemente a las funciones de agregación dentro del SELECT. De hecho, si se permite esta última forma es principalmente para ayudar a los desarrolladores que está habituados a escribir consultas en dialectos de SQL tradicionales.

Funciones matemáticas

eSQL ofrece un reducido número de funciones matemáticas predefinidas.

Función	Descripción
ABS	Devuelve el valor absoluto del valor recibido. Sintaxis: `ABS(<expresión>)`
CEILING	Devuelve el menor entero que no es menor que el valor recibido. Sintaxis: `CEILING(<expresión>)`
FLOOR	Devuelve el mayor entero que no es mayor que el valor recibido. Sintaxis: `FLOOR(<expresión>)`
ROUND	Redondea la parte entera del valor recibido al entero más cercano. Sintaxis: `ROUND(<expresión>)` Ejemplo: Para la expresión eSQL siguiente: `{ ROUND(2.8), CEILING(2.4), FLOOR(2.6), ABS(-1) }` se obtiene el conjunto de valores: `3.0`, `3.0`, `2.0` y `1.0`.

Funciones de manipulación de cadenas

El conjunto de funciones para la manipulación de cadenas de caracteres que ofrece eSQL es bastante completo, como podrá ver en la siguiente tabla.

Funciones	Descripción
Concat (cadena1, cadena2)	Devuelve una cadena resultado de la unión de cadena1 con cadena 2. Es análogo al operador + de ESQL

IndexOf(cadena1, cadena2)	Devuelve la posición de la cadena1 en la cadena2. El valor cero indica que no la ha encontrado, un valor superior índica el índice dónde comienza.
	`-- El siguiente ejemplo devuelve 4.`
	`IndexOf('xyz', 'abcxyz')`

Left (cadena, longitud)	Devuelve una cadena de la longitud dada comenzando por la izquierda de la cadena pasada como argumento.
	`-- El siguiente ejemplo devuelve abc.`
	`Left('abcxyz', 3)`

Length (cadena)	Devuelve la longitud en caracteres de la cadena
LTrim(cadena)	Devuelve una cadena sin espacios en blanco en su comienzo.
Replace (cadena1, cadena2, cadena3)	Devuelve la cadena1 con todas las apariciones de cadena2 reemplazadas por cadena3)

Reverse (cadena)	Devuelve una cadena con los caracteres en orden inverso
	`-- Por ejemplo, esta expresión devuelve dcba.`
	`Reverse('abcd')`

Right (cadena, longitud)	Devuelve una cadena de la longitud dada comenzando por la derecha (final) de la cadena pasada como argumento.
	`-- La siguiente expresión devuelve xyz.`
	`Right('abcxyz', 3)`

RTrim(cadena)	Elimina los espacios en blanco finales de la cadena.

Substring (cadena, comienzo, longitud)	Devuelve la subcadena de cadena comenzado en la posición dada por comienzo y con la longitud pasada. El índice de la cadena comienza en 1.
	`-- El siguiente ejemplo devuelve xyz.`
	`Substring('abcxyz', 4, 3)`

ToLower(cadena)	Devuelve la cadena con los caracteres todos en minúsculas
ToUpper(cadena)	Devuelve la cadena con los caracteres todos en mayúsculas

Trim(cadena)	Devuelve la cadena sin espacios en blanco al comienzo o al final.
	`-- El siguiente ejemplo devuelve 'a b c'.`
	`Trim('  a b c  ')`

Funciones de fecha y hora

Las funciones de fecha y hora se refieren siempre a la hora del servidor donde reside **el origen de datos**. Algunas funciones como `CurrentDateTimeOffset()` o `GetTotalMinutesOffset()` solo funcionan en SQL Server 2008 o superior.

Función	Descripción
CurrentDateTime()	Devuelve un valor de tipo System.Datetime con la fecha y hora en el servidor del almacen de datos.
CurrentDateTimeOffset()	Devuelve la fecha actual, la hora y el desplazamiento debido al uso horario como un valor del tipo **DateTimeOffset.** Actualmente solo funciona en servidores SqlServer 2008.

CurrentUtcDateTime()	Devuelve un valor System.DateTime con la fecha y hora actual en el servidor en uso horario universal

Day(*expresión* **)**	Devuelve el día del mes a partir de un valor DateTime o DateTimeOffset.
	`-- El siguiente ejemplo devuelve 03 en una cultura es-ES y 12 en en-US`
	`Day(cast('03/12/1998' AS DateTime))`

GetTotalOffsetMinutes (*DateTimeOffset* **)**	Devuelve el número de minutos que el uso horario local difiere de la hora universal (entre -780 y 780 minutos). Por ahora solo está disponible en SqlServer 2008

Hour (*expresión* **)**	Devuelve la hora de la expresión (DateTime, Time o DateTimeOffset) (entre 0 y 23)
	`-- El siguiente ejemplo devuelve 22.`
	`Hour(cast('22:35:5' AS DateTime))`

Millisecond(*expresión* **)**	Devuelve los milisegundos de la expresión (DateTime, Time o DateTimeOffset) (entre 0 y 999)

Minute(*expresión* **)**	Devuelve los minutos de la expresión (DateTime, Time o DateTimeOffset) (entre 0 y 59)
	`-- El siguiente ejemplo devuelve 35`
	`Minute(cast('22:35:5' AS DateTime))`

Month (*expresión* **)**	Devuelve los meses de la expresión (DateTime o DateTimeOffset) (entre 1 y 12)
	`-- El siguiente ejemplo devolvería 12 en una cultura es-ES y 3 en en-US`
	`Month(cast('03/12/1998' AS DateTime))`

Second(*expresión* **)**	Devuelve los segundos de la expresión (DateTime, Time o DateTimeOffset) (entre 0 y 59)
Year(*expresión* **)**	Devuelve el año de la expresión (DateTime o DateTimeOffset)

Operaciones entre bits

Las operaciones lógicas entre bits no suelen ser muy comunes, pero de todas formas eSQL ofrece el siguiente conjunto de funciones canónicas.

Operador	Descripción

BitWiseAnd (*valor1* **,** *valor2* **)**	Devuelve el resultado de hacer un and bit a bit entre valor1 y valor2.
	`-- El siguiente ejemplo devuelve 1.`
	`BitWiseAnd(1,3)`

BitWiseNot (*valor1* **)**	Devuelve valor1 con los bits negados.
	`-- El siguiente ejemplo devuelve -4.`
	`BitWiseNot(3)`

BitWiseOr (*valor1* **,** *valor2* **)**	Devuelve la operación lógica OR bit a bit entre valor1 y valor2
	`-- El siguiente ejemplo devuelve 3.`
	`BitWiseOr(1,3)`

BitWiseXor (*valor1* **,**	Devuelve la operación lógica XOR (OR exclusivo) bit a

valor2) bit entre valor1 y valor2.

```
-- El siguiente ejemplo devuelve 2.
BitWiseXor (1,3)
```

Otras funciones

La única función canónica que no entra en las demás categorías es `newGuid()` que crea un nuevo GUID. En caso del que el proveedor de acceso a datos sea SQL Server hace una llamada a la función T-SQL: `NEWID`.

APÉNDICE

ADO.NET Entity Framework 4.1: Algunas notas de rendimiento.

1.- INTRODUCCIÓN

Si bien Entity Framework 4.1 es un recurso muy potente en manos de desarrolladores, hay detalles que por mucho que la capa de abstracción de las tecnologías sea cada vez mayor, no podemos pasar por alto.

Este capítulo pretende ilustrar con ejemplos prácticos una serie de trucos y consejos de rendimiento a tener en cuenta cuando usamos Entity Framework 4.1.

Lo haremos de la siguiente manera, intentando ser lo más interactivo posible con el lector: establecido un determinado escenario para luego plantear una determinada solución.

Cada solución propuesta no tiene que ser vista como única, quizás hasta encuentre usted una mejor. Ese es precisamente el objetivo de este capítulo: motivarle a plantearse soluciones a escenarios reales de proyectos en los cuales puede usted aplicar algo de lo que hemos ido ilustrando a lo largo de este libro.

2.- CONSULTAS PARAMETRIZADAS

Empecemos por dos sencillas consultas *Linq*:

a)

```
List<Author>  result = (from a in context.Authors
                        where a.FirstName == "Pepe"
                        select a).ToList();
```

b)

```
string searchedName = "Pepe";
List<Author>  result = (from a in context.Authors
                        where a.FirstName == searchedName
                        select a).ToList();
```

Evidentemente si el valor de *searchName* es *"Pepe"* ambas consultas devuelven el mismo resultado. La pregunta entonces es: ¿dónde radica la problemática que hemos anunciado, y cuál es la diferencia entre ambas consultas?

El impacto que tiene cada una de estas consultas en el rendimiento, no es el mismo. Si bien es cierto que un ejemplo como este no sería determinante en el rendimiento de una aplicación, dadas las características de hardware con las que podemos contar hoy día, puede usted tomarlo como una especie de aperitivo dentro de este capítulo.

Para la primera consulta, sin parametrizar y con el valor de *searchName* como constante, Entity Framework genera una consulta AdHoc, ni siquiera DSE, implicando **que el plan de ejecución no se reaproveche**. Si esa misma consulta se realiza varias veces con valores de `FirstName` distintos al actual, se estará forzando un movimiento relativamente grande de elementos en la cache de planes de ejecución, lo cual conspiraría sin lugar a dudas en el rendimiento de la aplicación en cuestión.

Para el caso de la segunda consulta, Entity Framework la ejecuta mediante `sp_executesql (Force Statment Caching)`, por lo que esta consulta parametrizada puede reutilizar el mismo plan de ejecución, aun variando los valores de los parámetros tantas veces como sea necesario.

Para entender mejor las diferencias de estos comportamientos, puede usted ejecutar la siguiente consulta:

```
SELECT *
from sys.dm_exec_cached_plans
cp cross apply sys.dm_exec_sql_text(plan_handle)
```

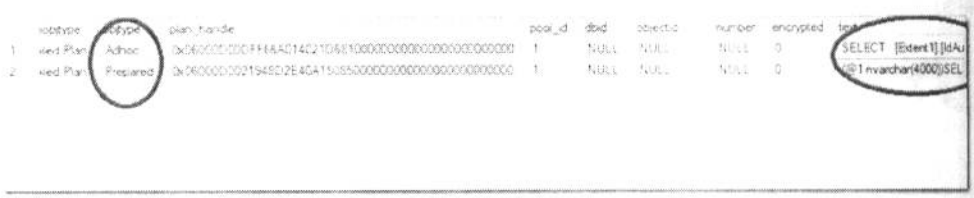


Figura C.1.- Diferencia entre consultas

La recomendación, por lo tanto para este caso, es evitar asignar valores constantes dentro de nuestras consultas y hacer uso de variables.

Desgraciadamente hay cierto tipo de consultas que no se comportan como quizás esperamos y aunque sigamos las recomendaciones antes expuestas podremos, si

revisamos nuestra cache de planes de ejecución, observar alguna consulta de tipo AdHoc.

Un caso concreto es el uso de los métodos extensores **SKIP** y **TAKE** de **L2E**, los cuales 'funcletizan' los valores y eliminan la posibilidad de hacer las consultas parametrizadas. Veamos un ejemplo para aclarar esto:

```
class Program
{
        static void Main(string[] args)
        {
            PagWithL2E(0, 10);
            PagWithESQL(0, 10);
        }
        static void PagWithL2E(int pageIndex, int pageCount)
        {
            int skip = pageIndex * pageCount;
            using (NLayerApp context = new NLayerApp())
            {
                List<Customer> customers =
                        context.Customers.OrderBy(c
                                                => c.CustomerCode)
                                                        .Skip(skip)
                                                .Take(pageCount)
                                                        .ToList();

            }
        }
        static void PagWithESQL(int pageIndex, int pageCount)
        {
            int skip = pageIndex * pageCount;
            using (NLayerApp context = new NLayerApp())
            {
                List<Customer> customers =
                     context.Customers.Skip("it.CustomerCode",
                                            "@skip",
                                             new
                                        ObjectParameter("skip",skip))
                                        .Top("@limit",
                                new ObjectParameter("limit", pageCount))
                                                        .ToList();

            }
        }
}
```

Ambos métodos obtienen una colección de *Customer* de forma paginada y hacen uso de variables para indicar los parámetros de paginación. Según lo expuesto anteriormente la consulta en L2E debería de ser parametrizada, ¿cierto?.

Sin embargo, un vistazo a los resultados haciendo una consulta sobre la vista administrada que nos da los elementos almacenados en la cache de planes de ejecución de SQL Server, demuestra lo contrario.

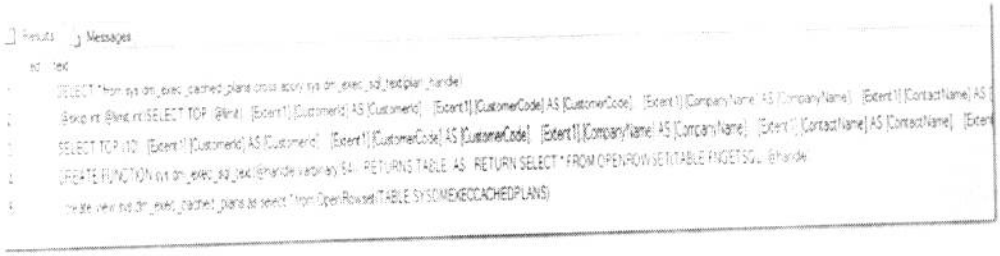


Figura C.2.- Cache de planes de Sql

La segunda fila que se corresponde en este caso a la consulta en **ESQL** ha quedado parametrizada, mientras que la consulta 3, **L2E**, es una consulta AdHoc.

Bien, ya hemos expuesto un problema concreto. Ahora toca presentar las soluciones, tal como adelantamos al inicio del capítulo.

Lógicamente ya se ha mencionado una posible: sustituir las consultas que usan paginación escritas en L2E por consultas que usen ESQL.

Otra solución es hacer uso de ***CompiledQuery*** tal y como ilustra el código a continuación.

```
class Program
    {
        static void Main(string[] args)
        {
            PagWithL2E(0, 10);
            PagWithESQL(0, 10);
        }
        static Func<NLayerApp,
                    int,
                    int,
                    IQueryable<Customer>> pagedQueryCompiled =

            CompiledQuery.Compile<NLayerApp,
                                  int,
                                  int,
                                  IQueryable<Customer>>((ctx,
                                                         skip,
                                                         pageCount)

             =>
            ctx.Customers.OrderBy(c => c.CustomerCode)
                                 .Skip(skip)
                                 .Take(pageCount));
        static void PagWithL2E(int pageIndex, int pageCount)
        {
            int skip = pageIndex * pageCount;
            using (NLayerApp context = new NLayerApp())
            {
                List<Customer> customers =
                                  pagedQueryCompiled.Invoke(context,
                                                            skip,
                                                            pageCount)
                                                           .ToList();

            }
        }
        static void PagWithESQL(int pageIndex, int pageCount)
        {
```

```
            int skip = pageIndex * pageCount;
            using (NLayerApp context = new NLayerApp())
            {
                List<Customer> customers =
                            context.Customers.Skip("it.CustomerCode",
                                                   "@skip",
                                    new ObjectParameter("skip",skip))
                                    .Top("@limit",
                             new ObjectParameter("limit", pageCount))
                             .ToList();
            }
        }
    }
```

Un examen a los planes de ejecución revela que ambas consultas son ahora parametrizadas, y por lo tanto nuestro *workaround* con *CompiledQuery* nos ha funcionado:

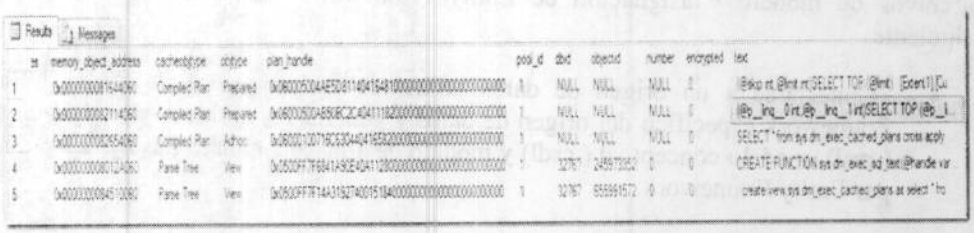


Figura C.3.- Cache de planes de Sql

3.- PRECOMPILACIÓN DE VISTAS

Uno de los elementos más fáciles de implementar y gracias al cual notaremos una importante mejora en el rendimiento de nuestras aplicaciones con ADO.NET Entity Framework 4, es el uso de la "*precompilación de vistas*". Uno de los puntos más significativos en la abstracción de la base de datos consiste precisamente en la creación de vistas de consulta y actualización en el lenguaje nativo de la base de datos.

Durante este proceso hay una carga importante de tiempo y de consumo de memoria (se aplica un caché dentro del dominio de aplicación en posteriores usos).

Con el fin de ilustrar cuán importante es este punto, en la siguiente gráfica, a modo de ejemplo se representa aproximadamente un 56% consumido por un proceso de ejecución de una consulta, estos datos se extraen directamente de la propia documentación de ADO.NET Entity Framework.

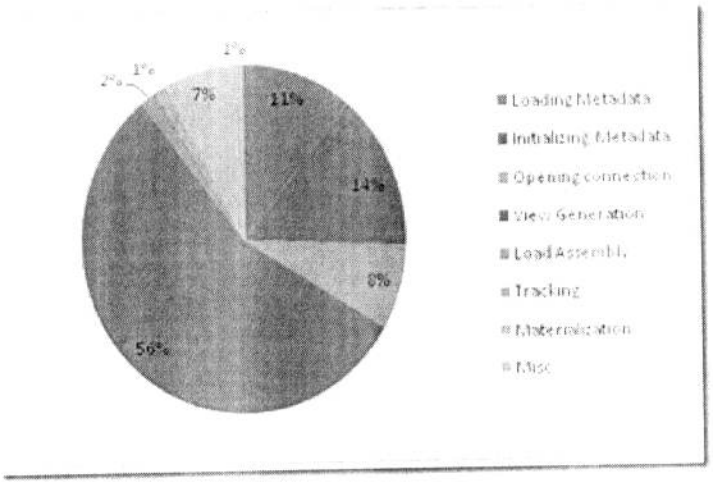


Figura C.4.- Ejecución de una consulta

Afortunadamente, haciendo uso de la herramienta **EDMGEN.exe,** podemos pre-compilar e incluir en el proyecto estas vistas, de tal forma que nos libremos en tiempo de ejecución de su creación.

EDMGEN.exe es una herramienta de línea de comandos utilizada para trabajar con archivos de modelo y asignación de Entity Framework. Puede utilizarse para lo siguiente:

- Conectarse a un origen de datos utilizando un proveedor de datos .NET Framework específico del origen de datos y generar los archivos de asignación (.msl), modelo conceptual (.csdl) y modelo de almacenamiento (.ssdl) utilizados por Entity Framework.
- Validar un modelo existente.
- Generar un archivo de código de C# o Visual Basic que contenga las clases de objetos generados a partir de un archivo de modelo conceptual (.csdl).
- Generar un archivo de código de C# o Visual Basic que contenga las vistas generadas previamente para un modelo existente.

Otra opción para pre-generar estas vistas es mediante la utilización de una simple plantilla T4, que realiza todo el trabajo de generación y nos libra de tener que incorporar opciones de pre-compilación a los proyectos que tengan modelos de Entity Framework. Un ejemplo de esta plantilla la puede encontrar dentro de los materiales de este libro en la ruta **tt\[NombreModelo].Views.tt**.

4.- LAZYLOADINGENABLED

La carga perezosa de forma implícita o automática con la que contamos en Entity Framework viene marcada de forma automática con la plantilla de generación de código por defecto, la plantilla de clases prescriptivas, por medio de una opción llamada `LazyLoadingEnabled`, tal como muestra el siguiente código y como hemos visto en el capítulo correspondiente. Note el constructor por defecto de un contexto de trabajo generado con ADO.NET EF.

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

Aunque esta opción es realmente cómoda (libra de tener que hacer un *Load* explícito de una propiedad de navegación), si no se es cuidadoso puede conspirar contra el rendimiento por un incremento innecesario de consultas a la base de datos.

Planteemos como escenario un ejemplo típico de una relación uno a muchos como la que se muestra en la siguiente figura.

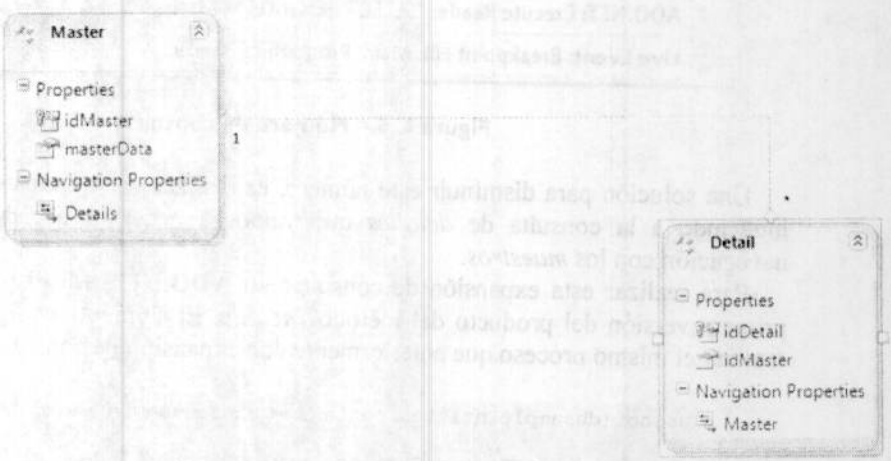


Figura C.5.- Relación uno a muchos

Partiendo de este modelo si se realiza la siguiente operación:

```
using (dbsampleEntities context = new dbsampleEntities())
{
       context.ContextOptions.LazyLoadingEnabled = true;

       foreach(Detail item in context.Details)
            Console.WriteLine(item.Master.masterData);
}
```

El número de consultas que mostraría cualquier profiler o el *Intellitrace* es N+1 siendo N el número de elementos de tipo *Detail* almacenados en la base de datos.

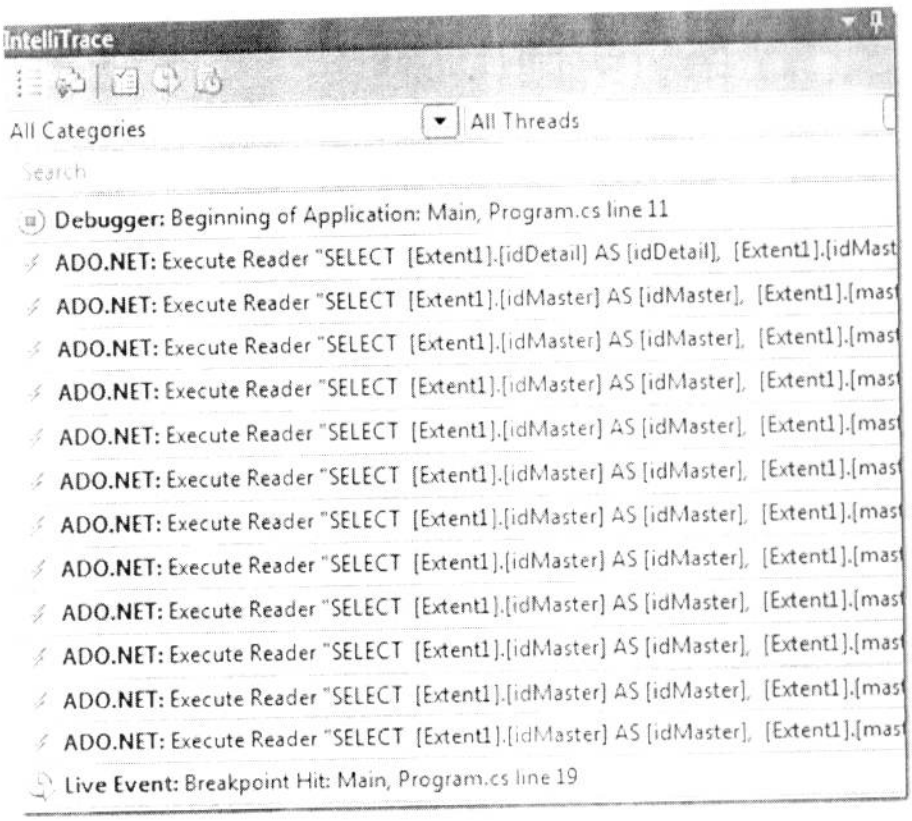


Figura C.6.- Número de consultas

Una solución para disminuir este número, es realizar una **expansión de consultas**, indicando a la consulta de *detalles* que vamos a necesitar acceder también a su navegación con los *maestros*.

Para realizar esta expansión de consultas en **ADO.NET EF** disponemos desde la primera versión del producto del método ***Include***. El siguiente código muestra cómo realizar el mismo proceso que anteriormente con expansión de consultas.

```
using (dbsampleEntities context = new dbsampleEntities())
{
      context.ContextOptions.LazyLoadingEnabled = true;

      foreach(Detail item in context.Details.Include("Master"))
           Console.WriteLine(item.Master.masterData);

      Console.ReadLine();
}
```

Ahora el número de las consultas se ha quedado solamente en 1, que consiste en un simple **INNER JOIN** entre *Detalle* y *Maestro*

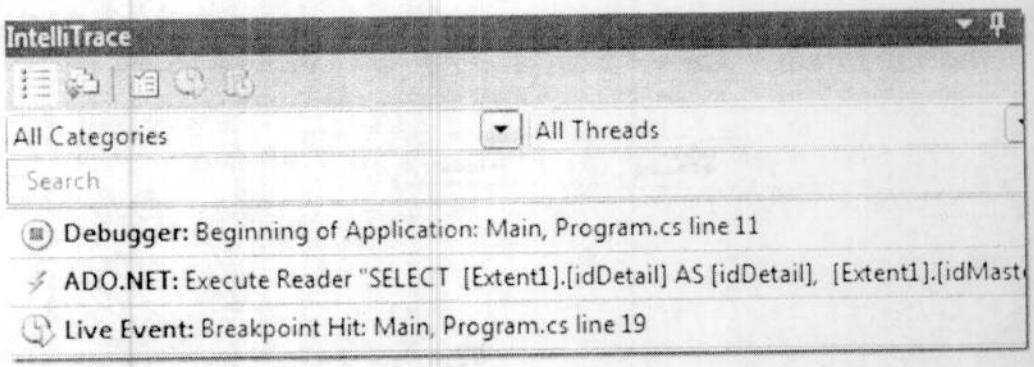


Figura C.7.- Número de consultas

5.- CAPACIDADES DE MODELADO

Tanto las antiguas como las nuevas capacidades de modelado nos permiten solventar la impedancia de nuestros dominios con respecto a los esquemas relacionales con nuestra base de datos. Pero muchas veces estos modelos contienen mucha profundidad de jerarquía y una gran cantidad de relaciones, de forma especial cuando la herencia es TPC-Style.

Para el ejemplo partiremos del siguiente modelo EDM sobre el cual vamos a ir realizando diferentes acciones para crear un modelo más adaptado a nuestro lenguaje.

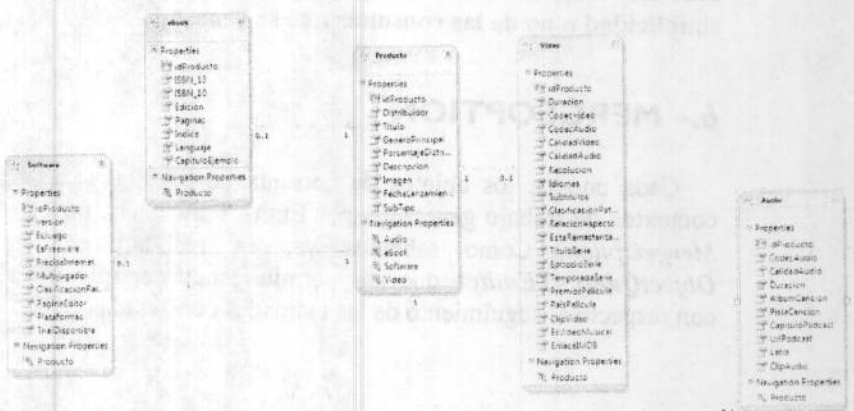


Figura C.8.- Modelo EDM

En este modelo se definen las entidades *Producto*, *Software*, *Audio*, *Ebook* y Video sobre las que vamos a trabajar y, empezar por ejemplo, a crear nuestra herencia de entidades, disponiendo de un solo *Producto* raíz.

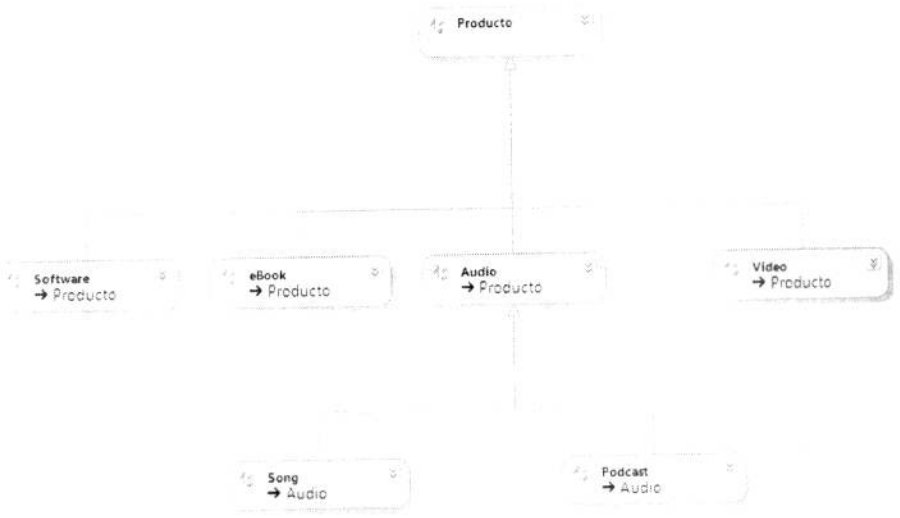


Figura C.9.- Modelo EDM

Desde luego este modelo es mucho más comprensible desde el punto de vista de un desarrollador Orientado a Objetos y seguramente tendrá más sentido para un dominio concreto en el que se encuentre, pero escenarios de modelado como este también suelen producir consultas excesivamente complejas que deberíamos analizar con criterio. Si dado este mecanismo de herencia TPC pidiéramos, por ejemplo, una lista de productos, tendríamos que valorar realmente qué es lo que estamos solicitando, y esto no es más claro que la unión de todas las entidades dentro de la jerarquía.

Lógicamente, en este tipo de escenarios no hay un *workaround* mágico que solucione las consultas complejas de un modelo, solamente podemos indicar que se debe ser muy cuidadoso con lo que se consulta y valorar **los modelos de EDM y la simplicidad o no de las consultas que se generan.**

6.- MERGEOPTION

Cada uno de los objetos de consulta de los que se dispone dentro de nuestros contextos de trabajo generados por Entity Framework poseen una propiedad llamada *MergeOption*. Como sabemos ya, en realidad es una propiedad de tipo ***ObjectQuery<TEntity>*** que nos permite establecer qué criterio deseamos establecer con respecto al seguimiento de las entidades consultadas.

Algunas de las opciones de este enumerado están indicadas para el tratamiento de concurrencia como `PreserveChanges` y `OverwriteChanges`, pero básicamente las opciones a discutir son si se realiza seguimiento de las entidades o no, `AppendOnly` o `NoTracking`.

En escenarios dónde los contextos tienen ciclos de vida cortos, el uso del seguimiento de las entidades no aporta más que una sobrecarga sobre el rendimiento debido a los distintos `ObjectStateEntry` generados.

Una recomendación fácil de implementar, es hacer que los distintos objetos de consulta (`ObjectSet` u `ObjectQuery`) tengan establecida la opción `NoTracking` en su propiedad **MergeOption** como podemos ver a continuación:

```
IObjectSet<TEntity> CreateObjectSet()
{
            if (_context != (IContext)null)
            {
                IObjectSet<TEntity> objectSet =
                                    _context.CreateObjectSet<TEntity>();

                ObjectQuery<TEntity> query =
                                     objectSet as ObjectQuery<TEntity>;

                if ( query != null )
                     query.MergeOption = MergeOption.NoTracking;

                return objectSet;
            }
            else
                throw new InvalidOperationException(
                 Resources.Messages.exception_ContainerCannotBeNull);
 }
```

7.- INDICES EN LAS CONSULTAS

Supongamos que tenemos una tabla con una columna de tipo *varchar*, es decir, una columna no Unicode, tal y como podría ser la siguiente:

```
   CREATE TABLE [dbo].[NonUnicodeTest](
    [idTest] [int] NOT NULL,
    [field] [varchar](10) NOT NULL,
CONSTRAINT [PK_NonUnicodeTest] PRIMARY KEY CLUSTERED
(
    [idTest] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
       IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON,
       ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
```

Al realizar una consulta con un filtro como `WHERE field=N' valor del campo'` podríamos tener un problema de NO uso de índices. Esta consulta no es *SARGABLE* (**http://en.wikipedia.org/wiki/Sargable**), existe una conversión de valor

no Unicode a valores Unicode. Lo lógico sería que Entity Framework se ocupe de este tema de forma implícita, pasemos a comprobarlo.

Si ejecutamos la siguiente consulta *Linq To Entities*;

```
using (EF4EXEntities context = new EF4EXEntities())
{
      string fieldValue ="field value";
      List<NonUnicodeTest> result =
                              (from nut in context.NonUnicodeTests
                                    where nut.field == fieldValue
                                    select nut).ToList();

       Console.ReadLine();
}
```

El SQL que llega a la base de datos es el siguiente:

```
exec sp_executesql N'SELECT
[Extent1].[idTest] AS [idTest],
[Extent1].[field] AS [field]
FROM [dbo].[NonUnicodeTest] AS [Extent1]
WHERE [Extent1].[field] = @p__linq__0',N'@p__linq__0
varchar(8000)',@p__linq__0='field value'
```

De momento todo parece indicar que ADO.NET EF resuelve este problema de forma correcta. Note que el campo que indica que sea Unicode o no es una atributo de la propiedad en el modelo conceptual, **CSDL**.

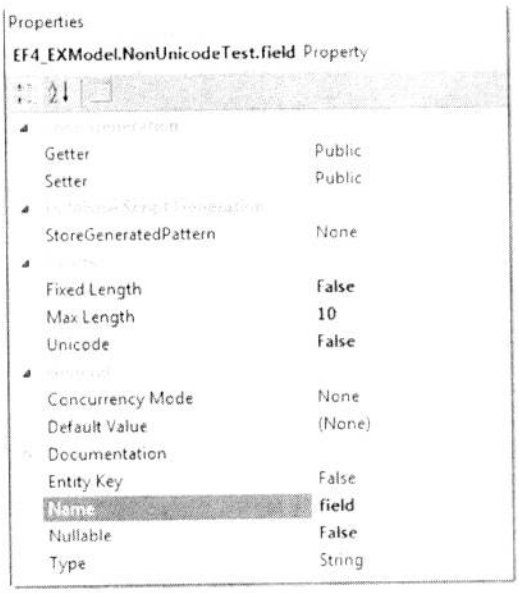


Figura C.10.- Propiedades de campo

Si realizamos las misma prueba con otras dos consultas adicionales:

```
using (EF4EXEntities context = new EF4EXEntities())
{
        string field1 = "field1";
        string field2 = "field2";

        List<NonUnicodeTest> result =
                    (from nut in context.NonUnicodeTests
                     where nut.field == field1 || nut.field == field2
                     select nut).ToList();

         Console.ReadLine();
 }

y

using (EF4EXEntities context = new EF4EXEntities())
{
        string field1 = "field1";
        string field2 = "field2";

        List<NonUnicodeTest> result =
                     (from nut in context.NonUnicodeTests
                     where nut.field == field1 && nut.field == field2
                     select nut).ToList();

         Console.ReadLine();
 }
```

Un examen a las trazas con un *profiler* de SQL para estas dos consultas delata los siguientes resultados, en el mismo orden en que se ilustran.

```
exec sp_executesql N'SELECT
[Extent1].[idTest] AS [idTest],
[Extent1].[field] AS [field]
FROM [dbo].[NonUnicodeTest] AS [Extent1]
WHERE [Extent1].[field] IN (@p__linq__0,@p__linq__1)',N'@p__linq__0
nvarchar(4000),@p__linq__1
nvarchar(4000)',@p__linq__0=N'field1',@p__linq__1=N'field2'

exec sp_executesql N'SELECT
[Extent1].[idTest] AS [idTest],
[Extent1].[field] AS [field]
FROM [dbo].[NonUnicodeTest] AS [Extent1]
WHERE ([Extent1].[field] = @p__linq__0) AND ([Extent1].[field] =
@p__linq__1)',N'@p__linq__0 varchar(8000),@p__linq__1
varchar(8000)',@p__linq__0='field1',@p__linq__1='field2'
```

Se han destacado las diferencias. En la sentencia OR todo parece indicar que Entity Framework no es capaz de reconocer el patrón de *unicode/no unicode*.

Una forma de resolver esta situación es usando una *Model Defined Function*. De la misma forma que podemos crear nuevas funciones definidas en el modelo, EF pone a nuestra disposición un conjunto de ellas "de serie".

Estas funciones las podemos encontrar dentro de las clases `EntityFunctions` y `SqlFunctions`, en concreto `EntityFunctions` brinda una función definida en el modelo con el nombre `AsNonUnicode`.

```
using (EF4EXEntities context = new EF4EXEntities())
{
        string field1 = "field1";
        string field2 = "field2";

        List<NonUnicodeTest> result =
              (from nut in context.NonUnicodeTests
              where nut.field == EntityFunctions.AsNonUnicode(field1)
              ||
              nut.field == EntityFunctions.AsNonUnicode(field2)
              select nut).ToList();

       Console.ReadLine();
}
```

El resultado, con el uso de `EntityFunctions` ya se asemeja más a lo que esperamos:

```
exec sp_executesql N'SELECT
[Extent1].[idTest] AS [idTest],
[Extent1].[field] AS [field]
FROM [dbo].[NonUnicodeTest] AS [Extent1]
WHERE [Extent1].[field] IN (@p__linq__0,@p__linq__1)',N'@p__linq__0
varchar(8000),@p__linq__1
varchar(8000)',@p__linq__0='field1',@p__linq__1='field2'
```

8.- CONCLUSIÓN

La idea con este apéndice no es más que ilustrar mediante ejemplos prácticos algunos escenarios que impliquen problemas de rendimiento e intentar arrojar luz sobre posibles soluciones. Este debería ser un apéndice abierto, que esperamos que el lector más interesado convierta en la introducción de su propia lista de trucos de rendimiento para Entity Framework 4.1.

APÉNDICE

Plantillas T4

1.- INTRODUCCIÓN

Text Template Transformation Toolkit (T4) es un conjunto de herramientas/recursos para la generación de código basado en plantillas. Se pueden usar plantillas T4 para generar código Visual Basic, C#, T-SQL o XML entre otros. Una plantilla T4 es una combinación de bloques de texto y lógica expresada en algún lenguaje .Net y generada como un archivo de texto. La lógica es descrita en forma de fragmentos de código C# o Visual Basic. El archivo generado puede ser texto de cualquier tipo, como una página web o un archivo de recursos, así como código fuente. Las plantillas de texto pueden usarse en tiempo de ejecución para generar parte del resultado de una aplicación, así como para la generación de código de forma dinámica.

Las plantillas T4 tienen una sintaxis similar a la que se usa en ASP.NET: directivas de procesamiento, bloques de texto y bloques de código:

```
<#@ template language="C#" #>
Hola <# Write("T4!") #>
```

Con el uso de la directiva de procesamiento `<#@ template #>` se especifica que el código funcional de la plantilla en cuestión se escribe en C #.

Para generar una salida determinada a partir de una plantilla T4 los pasos son los que se ilustran a continuación.



Figura D.1.- Generando desde una plantilla T4

En el primer paso la plantilla es "*compilada*": se *parsean* las instrucciones, bloques de texto y de código, generándose una clase concreta `TextTransformation`, que se compilará como contenido de un ensamblado.NET. Como segundo paso del proceso se crea una instancia de la clase generada, y se invoca su método `TransformText` retornando como salida el `string` que este devuelve.

Si pudiéramos generar nuestro libro con una plantilla T4, seguramente cumpliendo con estándares, el primer ejemplo que presentaríamos sería un "*Hola Mundo*":

```
<#@ template language="C#" #>
using System;

public class <#= this.ClassName #>
{
   public static void HelloWorld()
   {
      Console.WriteLine("Hola Mundo");
   }
}
<#+
string ClassName = "FirstDemo";
#>
```

Para una plantilla como la anterior se genera una clase con el nombre especificado en `ClassName`. Cuando guardemos el fichero .tt, Visual Studio automáticamente genera el .cs correspondiente:

```
using System;

public class FirstDemo
{
    public static void HelloWorld()
    {
        Console.WriteLine("Hola Mundo");
    }
}
```

Cada vez que se realice alguna modificación en la plantilla y se guarden las modificaciones sobre la misma (equivalente a la opción del menú de contexto sobre el fichero .tt *"Run Custom Tool"*) se regenera el código de la clase correspondiente (`FirstDemo`).

2.- LAS PLANTILLAS T4

Cada plantilla es una mezcla del texto tal y como aparecerá en el archivo generado, y de fragmentos de código. Los fragmentos de código por regla general definen valores para las partes variables del archivo.

Por ejemplo, puede generarse un HTML que contenga algunas tablas de valores mostrado dentro de un formato fijo que contenga algún texto estático. Puede comenzar el diseño escribiendo un prototipo de la página HTML y, a continuación, reemplazar la tabla y otras partes variables con código que genere el contenido dinámico. Esta estructura permite tener una aproximación de la forma final de la salida de un plantilla (la parte estática del supuesto HTM por ejemplo), haciendo que cualquier cambio en el formato de la salida resulte más sencillo y confiable.

Las plantillas pueden dividirse en dos grupos: las plantillas de texto y las plantillas de texto pre-procesadas. La diferencia fundamental radica en la *"Custom tool"* que las interpreta (Figura 2) y por ende las diferencias y dependencias en el código que para éstas se genera.



Figura D.2.- Plantilla de texto Custom Tool (TextTemplatingFileGenerator)

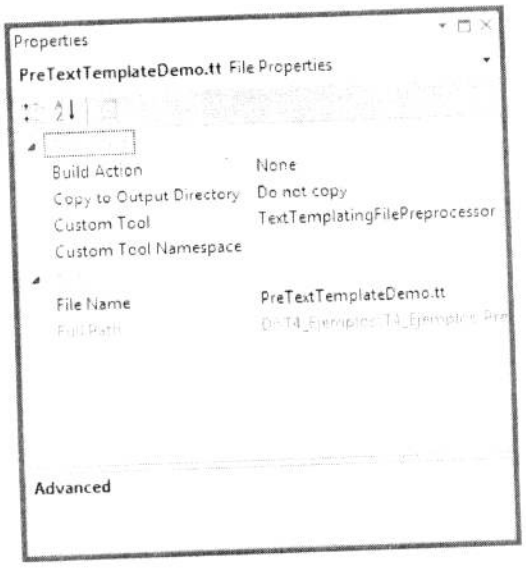


Figura D.3.- Plantilla pre-procesada Custom Tool(TextTemplatingFilePreprocessor)

La estructura de una plantilla consiste en un conjunto directivas, cuya sintaxis, por regla general, es la siguiente:

```
<#@ DirectiveName [ParameterName = "ParameterValue"] #>
```

Componiendo bien bloques de texto como bloques de código que forman el cuerpo de una plantilla T4.

- **Bloques de texto**: contenido que se copia directamente en el resultado.

Un bloque de texto inserta texto directamente en el archivo de salida. No existe ningún formato especial para los bloques de texto. Por ejemplo, la siguiente plantilla de texto generará un archivo de texto con la palabra `"Hola T4!"`:

```
<#@ output extension=".txt" #>
Hola T4!
```

- **Bloques de código**: código que inserta valores variables en el texto y controla las partes condicionales o repetidas del mismo.

2.1.- Bloques de código

Los bloques de código son, como su propio nombre indica, secciones de código que se utilizan para transformar las plantillas. El lenguaje predeterminado es C#, para utilizar, por ejemplo Visual Basic puede escribir esta directiva al principio del archivo:

```
<#@ template language="VB" #>
```

El lenguaje en el que escribe el código de los bloques no está relacionado con el lenguaje del texto que se genera. Puede mezclar cualquier número de bloques de texto y bloques de código en un archivo de plantilla. Sin embargo, no puede colocar un bloque de código dentro de otro. Cada bloque de código se delimita con los símbolos `<# ... #>`. Puede insertar un bloque de texto en cualquier posición del código donde se podría incluir una instrucción tipo `Write();`.

2.1.1.- Bloques de control de expresiones

Un bloque de control de expresiones da como resultado una expresión y la convierte en una cadena. Esta cadena se inserta en el archivo de salida.

Los bloques de control de expresiones se delimitan con los símbolos `<#= ... #>`. Por ejemplo, el siguiente bloque de control hace que el archivo de salida contenga `"5"`:

```
<#= 2 + 3 #>
```

Observe que el símbolo de apertura tiene los caracteres `"<#="`. La expresión puede incluir cualquier variable que esté en el ámbito. Por ejemplo:

```
<#@ output extension=".txt" #>
<#
    for(int i = 0; i < 4; i++)
    {
     #>
     Hola número <#= i+1 #>:!<#
    }
#>
```

2.1.2.- Bloques de control de características de clase

Un bloque de control de características de clase define propiedades, métodos o cualquier otro código que no se debe incluye en la transformación principal. Los bloques de características de clase se utilizan con frecuencia para definir funcionalidades auxiliares y suelen almacenarse en archivos independientes para que más de una plantilla de texto pueda incluirlos.

Los bloques de control de características de clase se delimitan con los símbolos `<#+ ... #>`. Por ejemplo, el siguiente archivo de plantilla declara y utiliza un método:

```
<#@ output extension=".txt" #>
<#
    for(int i = 0; i < 4; i++)
    {
#>
    El cuadrado de <#= i #> es <#= Square(i+1) #>.<#
    }
#>
<#+
private int Square(int i)
{
    return i*i;
}
#>
```

Las características de clase se deben colocar al final del archivo donde se definen. Con la instrucción `<#@include#>` se incluye una característica de clase, aunque la directiva `include` vaya seguida de bloques estándar y texto.

Los bloques de características de clase pueden contener bloques de texto. Por ejemplo:

```
<#
   for(int i = 0; i < 4; i++)
   {  WriteSquareLine(i); }
#>
<#+
private void WriteSquareLine(int i)
{
#>
   El cuadrado de <#= i #> es <#= i*i #>.<#
}
#>
```

Resulta especialmente útil para definir estos métodos que describen una determinada funcionalidad a usar en la plantilla en archivo independiente, permitiendo de esta forma que puede incluirse en más de una plantilla.

2.1.3.- Utilizar definiciones externas

Ensamblados

Los bloques de código de la plantilla pueden utilizar tipos definidos en los ensamblados de uso más frecuente de .NET, como `System.dll`. Además, puede hacerse referencia a otros ensamblados .NET, proporcionando el nombre del ensamblado:

```
<#@ assembly name="System.Xml.dll" #>
```

La directiva `assembly` no tiene ningún efecto en una plantilla de texto preprocesada. Tiene un parámetro `name` que identifica el ensamblado al que se va a hacer referencia en los bloques de código de la plantilla de texto. Esto permite utilizar tipos dentro de ese ensamblado desde el código de la plantilla de texto. El nombre del ensamblado debe ser uno de los siguientes:

- El nombre de archivo del ensamblado, si se encuentra en el mismo directorio que la plantilla de texto
- La ruta de acceso absoluta del ensamblado
- La ruta de acceso relativa del ensamblado (con respecto al directorio donde se encuentra la plantilla de texto)
- El nombre seguro del ensamblado, como en la GAC
- También puede utilizar la sintaxis $(variableName) para hacer referencia a las variables de entorno y las variables de MSBuild o Visual Studio.

Espacios de nombre

La directiva `import` tiene un parámetro `namespace` que permite hacer referencia a los elementos de un ensamblado al que se referencia sin proporcionar un nombre completo en el código que se agrega a una plantilla de texto. Equivale a `using` en C# o a `imports` en Visual Basic.

```
<#@ import namespace="System.Xml" #>
```

Puede utilizar tantas directivas `assembly` e `import` como desee. Debe colocarlas antes de los bloques del texto y de control donde se utilicen.

Incluyendo código y texto

La directiva `include` inserta texto de otro archivo de plantilla. Por ejemplo, esta directiva inserta el contenido de test.txt.

```
<#@ include file="c:\test.txt" #>
```

El contenido incluido se procesa casi como si formase parte de la plantilla de texto que lo incluye. Puede también incluir un archivo que contenga un bloque de características de clase `<#+...#>` aunque la directiva de inclusión vaya seguida de texto normal y de bloques de control estándar.

2.1.4.- Utility Methods

Existen varios métodos como `Write()` que siempre están disponibles en un bloque de control. Incluyen métodos para ayudarle a aplicar sangría al resultado y para notificar errores. También puede escribir su propio conjunto de métodos de utilidad.

Para especificar parámetros en la transformación que lleva a cabo una plantilla durante la generación de una salida determinada, contamos con la directiva template. Esta consta de varios parámetros que permiten especificar diferentes aspectos de la transformación. Todos los parámetros son opcionales.

Tabla 1.- Directivas de Plantilla

Nombre del parámetro	Valor predeterminado	Valores aceptados
language	C#	C#, VB
inherits	TextTransformation	Cualquier clase heredera de TextTransformation
culture	""	Cualquier string que represente una cultura en el formato "xx-XX"
debug	true	true, false
hostspecific	true	true, false

Con los valores de estos parámetros se especifica el lenguaje (Visual Basic o C#) que se va a usar para el código fuente en los bloques de instrucciones y expresiones. Todo el código generado en la clase de transformación utilizará este lenguaje.

También podemos definir la clase base de la clase de transformación generada con el parámetro `inherits`. Una cultura determinada que se va a utilizar cuando un bloque de expresiones se convierte en texto (`System.Globalization.CultureInfo`, así como habilitar o no la depuración. Si se habilita la depuración, el archivo generado se escribirá en el directorio %TEMP% y podrá depurar el código de una plantilla de texto.

2.1.5.- Directivas de Output

La directiva output tiene parámetros que permiten especificar características de la salida de texto generada, como la extensión de nombre de archivo. Todos los parámetros son opcionales.

La directiva de salida no se requiere en una plantilla de texto pre-procesada. En su lugar, la aplicación obtiene una cadena mediante una llamada a `TextTransform()`.

Tabla 2.- Directivas de Salida

Nombre del parámetro	Valor predeterminado	Valores aceptados
extension	.cs	Cualquier cadena que satisfaga las reglas del sistema de archivos para una extensión de nombre de archivo.
encoding	La codificación utilizada por el archivo de plantilla de texto.	ASCII, UNICODE, UTF-8, UTF-7, UTF-32. En general, cualquiera de las codificaciones que devuelve EncodingGetEncoding().

Podemos definir la extensión del archivo de salida de texto generado. Por ejemplo, puede especificar una extensión .cs o .vb si el texto de salida generado es un archivo de código. Es importante recordar que la extensión de nombre de archivo solamente indica que el archivo está previsto para un formato determinado, esto no controla de ninguna forma que el texto generado cumple los requisitos del formato deseado (es responsabilidad de quién define la plantilla).

```
<#@ output extension=".txt" #>
<#@ output extension=".htm" #>
<#@ output extension=".cs" #>
<#@ output extension=".vb" #>
```

3.- PLANTILLAS VS PLANTILLAS PRE-PROCESADAS

Las plantillas (nos referiremos así a las no pre-procesadas) tienen una dependencia de ensamblado `Microsoft.VisualStudio.TextTemplating`, que se instala como parte de Visual Studio. Por su parte las plantillas T4 pre-procesadas no contienen ninguna dependencia y pueden utilizarse desde prácticamente cualquiera de nuestras aplicaciones.

Para el caso de una plantilla T4 pre-procesada todo el código necesario para "ejecutar" la plantilla es generado como parte de su propia definición. De cualquier forma una buena práctica es evitar estas definiciones redundantes para cada plantilla

dentro de nuestros proyectos definiendo una herencia con el parámetro `Inherits` en la directiva `Template` para de esta forma especificar la clase `TextTransformation` como clase base de nuestras plantillas, haciendo el código de las mismas más legible y facilitando el mantenimiento de las mismas.

A modo de directriz general, es preferible no escribir cuerpos grandes de código en las plantillas. En su lugar, defina métodos en una clase parcial y llame a los métodos desde la plantilla. De esta manera, la plantilla muestra más claramente el aspecto que tendrá el archivo de salida de destino, y el análisis de este aspecto se podrá separar de la lógica de creación de los datos que se muestran.

El contenido incluido puede contener cualquier mezcla de código de programa y texto sin formato, y puede contener otras directivas de inclusión.

La directiva de inclusión se puede utilizar en cualquier parte de un archivo de plantilla o de un archivo de texto incluido.

3.1.- Plantillas pre-procesadas

A partir de ahora nos centraremos en las plantillas pre-procesadas, siendo las que más se utilizan en este libro. Anteriormente ilustramos una serie de pasos por los que transita la plantilla T4 durante la generación de código a partir de la misma. Visual Studio 2010 también permite pre-procesar una plantilla en tiempo de diseño, cuando se crea la propia plantilla. En ejecución, cuando un desarrollador usa la plantilla para generar cierto código, la aplicación (la que contendrá dicho código generado), crea una instancia de la previamente compilada instancia de `GeneratedTextTransformation` y a partir de esta se generará la salida de la T4.

Para crear una plantilla en el Explorador de soluciones, haga clic con el botón secundario en el proyecto, elija **Agregar** y haga clic en **Nuevo elemento**.

1. En el cuadro de diálogo Agregar nuevo elemento, seleccione Plantilla de texto preprocesada.
2. Escriba un nombre para el archivo de plantilla.
3. Haga clic en Agregar.

Se crea un nuevo archivo con la extensión **.tt**. Su propiedad **Herramienta personalizada** se establece en **TextTemplatingFilePreprocessor**.

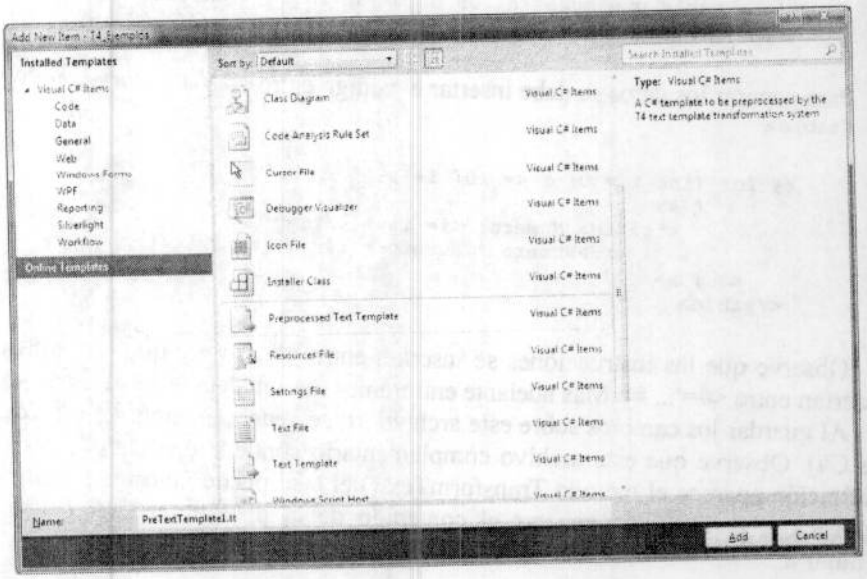


Figura D.4.-

Una buena manera de crear una plantilla es convertir un archivo existente en una plantilla. Por ejemplo, si la aplicación generará archivos HTML, puede comenzar creando un archivo HTML sin formato. Asegúrese de que funciona correctamente y de que su aspecto es correcto. A continuación, inclúyalo en el proyecto de Visual Studio 2010 y conviértalo en una plantilla. Para convertir un archivo de texto existente en una plantilla.

1. Incluya el archivo en el proyecto, haga clic con el botón secundario del ratón, elija **Agregar** y, a continuación, haga clic en **Elemento existente**.

2. Establezca la propiedad **Herramienta personalizada** del archivo en **TextTemplatingFilePreprocessor**. Si la propiedad tenía valor, asegúrese de que es **TextTemplatingFilePreprocessor** y no **TextTemplatingFileGenerator** (sucede si el archivo tiene extensión **.tt**.) Opcionalmente podemos cambiar la extensión **.tt** para evitar abrir el archivo en un editor incorrecto.

3. Insertar la siguiente línea al principio del archivo. `<#@ template language="C#" #>`

Como ya hemos mencionado anteriormente la primera línea de la plantilla indica el lenguaje .Net con el que se define la lógica dentro de la plantilla como se indica a continuación:

```
<#@ template language="C#" #>
```

El parámetro de lenguaje dependerá del lenguaje del proyecto. Luego se puede incluir dentro del archivo **.tt** el texto correspondiente que queremos genere la aplicación. Por ejemplo:

```
<html><body>
<h1>Un número y su cuadrado</h1>
<!—La tabla que muestra los datos se incluye en este espacio -->
</body></html>
```

Para generar los datos se debe insertar el código entre <# y #>. Por ejemplo:

```
<table>

<# for (int i = 0; i <= 10; i++)
       { #>
         <tr><td> Número: <#= i #> </td>
             <td>Número * Número = <#= i * i #> </td> </tr>
    <# } #>
 </table>
```

Observe que las instrucciones se insertan entre <# ... #> y que las expresiones se insertan entre <#= ... #>.Más adelante entraremos más en detalle sobre esta sintaxis.

Al guardar los cambios sobre este archivo **.tt**, se genera un archivo **.cs**. (para el caso de C#). Observe que este archivo complementario contiene una clase parcial en cuya definición aparece el método TransformText() el cuál puede invocarse desde la propia aplicación, permitiendo generar el contenido de la plantilla mediante una llamada similar a:

```
MyWebPage page = new MyWebPage();
String pageContent = page.TransformText();
System.IO.File.WriteAllText("outputPage.html", pageContent);
```

Normalmente una plantilla debe importar algunos datos de otras partes de la aplicación. Para que el proceso de mantenimiento resulte más sencillo y organizado, el código que compila la plantilla es una clase parcial. Puede crear otra parte de la misma clase en otro archivo del proyecto. Este archivo puede incluir un constructor con parámetros, propiedades y funciones a los que pueden tener acceso el código incrustado en la plantilla y el resto de la aplicación.

Por ejemplo, para un archivo independiente **MyWebPageCode.cs**:

```
partial class MyWebPage
{
    private MyData m_data;
    public MyWebPage(MyData data) { this.m_data = data; }
}
```

En el archivo de plantilla **MyWebPage.tt**, podría escribir:

```
<h2>Sales figures</h2>
<table>
```

```
<# foreach (MyDataItem item in m_data.Items)
   // m_data en MyWebPageCode.cs
   { #>
      <tr><td> <#= item.Name #> </td>
          <td> <#= item.Value #> </td></tr>
<# }
#>
</table>
```

Para utilizar esta plantilla en la aplicación:

```
MyData data = ...;
MyWebPage page = new MyWebPage(data);
String pageContent = page.TransformText();
System.IO.File.WriteAllText("outputPage.html", pageContent);
```

Un método alternativo para pasar datos a la plantilla consiste en agregar propiedades públicas a la clase de plantilla, que la aplicación puede establecer antes de llamar a `TransformText()`.

4.- TRANSFORMAR DATOS Y MODELO

Las plantillas de texto permiten generar texto plano, código, archivos de configuración y otros archivos a partir de un modelo determinado. Nos referimos en este caso a modelo como un archivo o una base de datos que contiene la información esencial que define una parte determinada de la aplicación. Por ejemplo, puede tener un modelo que defina un flujo de trabajo como un diagrama de flujo. Desde el modelo, puede generar el código que ejecuta el flujo de trabajo.

Probablemente ya está familiarizado con la generación de código. Por ejemplo al definir los recursos en un archivo .resx para un proyecto determinado en Visual Studio, se genera automáticamente un conjunto de clases y métodos .El archivo de recursos hace que resulte más fácil y confiable editar los recursos de lo que resultaría editar las clases y los métodos. Con las plantillas de texto, puede generar código de la misma manera desde un origen con un diseño propio. Normalmente, se generan varios archivos de una solución de Visual Studio con un único modelo de entrada. Cada archivo se genera a partir de su propia plantilla, pero todas las plantillas hacen referencia al mismo modelo. Si el modelo de origen cambia, debe volver a ejecutar todas las plantillas de la solución. Para ello, haga clic en **Transformar todas las plantillas** en la barra de herramientas del Explorador de soluciones.

La aplicación más útil para una plantilla de texto consiste en generar material basado en el contenido de un origen, como una base de datos o un archivo de datos. La plantilla extrae y cambia el formato de los datos. Una colección de plantillas puede transformar este origen en varios archivos.

Hay varios enfoques fundamentales para leer el archivo de origen.

- **Leer un archivo en la plantilla de texto**. Esta es la manera más sencilla de incluir los datos en la plantilla:

```
<#@ import namespace="System.IO" #>
<# string fileContent = File.ReadAllText("myData.txt"); ...
```

- **Cargar un archivo como un modelo navegable**. Un método más eficaz consiste en leer los datos como un modelo, en el que el código de la plantilla de texto puede navegar. Por ejemplo, puede cargar un archivo XML y navegar en él con expresiones XPath.

- **Editar el archivo de modelo en un diagrama o formulario.** Domain-Specific Language Tools proporciona herramientas que permiten editar un modelo como un diagrama o un formulario Windows Forms. Así resulta más fácil analizar el modelo con los usuarios de la aplicación generada. Domain-Specific Language Tools también crea un conjunto de clases fuertemente tipadas que reflejan la estructura del modelo.

- **Utilizar un modelo UML**. Puede generar código a partir de un modelo UML. Esto ofrece la ventaja de que el modelo se puede editar como un diagrama en una notación familiar. Además, no tiene que diseñar el diagrama.

Las plantillas T4, integradas en el entorno de desarrollo y de sintaxis muy intuitiva, constituyen un recurso perfecto para automatizar parte del desarrollo de software.

APÉNDICE

EDMGen

1.- INTRODUCCIÓN

Este apéndice lista las opciones que ofrece la herramienta de línea de comandos **EDMGen.exe**.

La sintaxis necesaria para utilizar el comando es:

```
EdmGen /mode:<modo> [ <opciones> ]
```

en donde `<modo>` debe ser uno de los siguientes:

Modo	Descripción
/mode:ValidateArtifacts	Valida los ficheros **.csdl**, **.ssdl** y **.msl**, y muestra los errores o advertencias que hubieran. Esta opción requiere al menos uno de los argumentos **/inssdl** o **/incsdl**. Si se especifica **/inmsl**, también se necesitan los argumentos **/inssdl** y **/incsdl**.
/mode:FullGeneration	Utiliza la información de conexión a base de datos especificada en el argumento **/connectionstring** y genera los ficheros **.csdl**, **.ssdl**, **.msl**, el fichero de código fuente para la capa de objetos y el fichero de vistas. Esta opción requiere un argumento **/connectionstring** y un argumento **/project**, o en su lugar los argumentos **/outssdl**, **/outcsdl**, **/outmsdl**, **/outobjectlayer**, **/outviews**, **/namespace** y **/entitycontainer**.
/mode:FromSSDLGeneration	Genera los ficheros **.csdl** y **.msl**, el fichero de código fuente y el fichero de vistas a partir del

	fichero **.ssdl** especificado. Esta opción requiere el argumento **/inssdl** y un argumento **/project**, o en su lugar los argumentos **/outcsdl**, **/outmsl**, **/outobjectlayer**, **/outviews**, **/namespace** y **/entitycontainer**.
/mode:EntityClassGeneration	Crea un fichero de código fuente que contiene las clases generadas a partir del fichero **.csdl**. Esta opción requiere el argumento **/incsdl** y el argumento **/project** o el argumento **/outobjectlayer**. El argumento **/language** es opcional.
/mode:ViewGeneration	Crea un fichero de código fuente que contiene las vistas generadas a partir de los ficheros **.csdl**, **.ssdl** y **.msl**. Esta opción requiere los argumentos **/inssdl**, **/incsdl** e **/inmsl**, y el argumento **/project**, o en su lugar el argumento **/outviews**. El argumento **/language** es opcional.

Las `<opciones>` son las siguientes:

Opción	Descripción
/p[roject]:<cadena>	Especifica el nombre de proyecto a utilizar. El nombre de proyecto se utiliza como valor por defecto para el espacio de nombres, el nombre de los ficheros de EDM, el nombre del fichero de código fuente con las clases generadas y el nombre del fichero de código fuente con las vistas generadas. El nombre del contenedor de entidades será **<proyecto>Context.**
/prov[ider]:<cadena>	El nombre del proveedor de datos de .NET Framework que se utilizará para generar el fichero de modelo de almacén (**.ssdl**). El proveedor predeterminado es el de SQL Server (System.Data.SqlClient).
/c[onnectionstring]:<cadena de conexión >	Especifica la cadena de conexión a utilizar para conectar con la base de datos.
/incsdl:<fichero>	Especifica el fichero **.csdl** o un directorio donde están situados los ficheros **.csdl**. Este argumento puede ser usado múltiples veces, de modo que se pueda especificar varios directorios o ficheros **.csdl**. Especificar múltiples directorios puede ser útil para la generación de clases (**/mode:EntityClassGeneration**) o de

	vistas (**/mode:ViewGeneration**) cuando el modelo conceptual está dividido en múltiples ficheros. También puede ser útil si se desea validar múltiples modelos (**/mode:ValidateArtifacts**).
/refcsdl:<fichero>	Especifica el fichero o ficheros **.csdl** adicionales a utilizar para resolver cualquier referencia en el fichero fuente **.csdl** (el especificado mediante la opción **/incsdl**). El fichero **/refcsdl** contiene tipos de los que depende el fichero fuente **.csdl**. Este argumento puede ser usado múltiples veces.
/inmsl:<fichero>	Especifica el fichero **.msl** o un directorio donde están situados los ficheros **.msl**. Este argumento puede ser usado múltiples veces, de modo que se pueda especificar varios directorios o ficheros **.msl**. Especificar múltiples directorios puede ser útil para la generación de clases (**/mode:EntityClassGeneration**) o de vistas (**/mode:ViewGeneration**) cuando el modelo conceptual está dividido en múltiples ficheros. También puede ser útil si se desea validar múltiples modelos (**/mode:ValidateArtifacts**).
/inssdl:<fichero>	Especifica el fichero **.ssdl** o un directorio donde está situado el fichero **.ssdl**.
/outcsdl:<fichero>	Especifica el nombre del fichero **.csdl** que será creado.
/outmsl:<fichero>	Especifica el nombre del fichero **.msl** que será creado.
/outssdl:<fichero>	Especifica el nombre del fichero **.ssdl** que será creado.
/outobjectlayer:<fichero>	Especifica el nombre del fichero de código fuente que contendrá los objetos generados a partir del fichero **.csdl**.
/outviews:<fichero>	Especifica el nombre del fichero de código fuente que contendrá las vistas generadas.
/language:[VB|CSharp]	Especifica el lenguaje de programación para los ficheros de código fuente. Por defecto es C#.
/namespace:<cadena>	Especifica el espacio de nombres a utilizar. El espacio de nombres se asigna en el fichero **.csdl** cuando se utiliza uno de los modos **/mode:FullGeneration** o **/mode:FromSSDLGeneration**. El espacio de nombres no se utiliza al ejecutar el

	modo **/mode:EntityClassGeneration**.
/entitycontainer:<cadena>	Especifica el nombre a aplicar al elemento `<EntityContainer>` en los ficheros EDM generados.
/help o /?	Muestra la sintaxis de la aplicación y las opciones disponibles.
/nologo	Suprime la visualización del mensaje de *copyright*.

Índice analítico

A

Actualización de entidades, 208
ADO.NET Entity Framework, 5
aggregate root, 331
Aggregate Root, 24
agregado de entidades, 24
agrupación, 375
anemic domain model, 331
Anti-corruption layer, 95
API fluent, 280, 283, 285, 286, 287, 288, 292, 294, 301, 307, 311, 328, 333
ApplyChanges, 217
asistente de actualización de modelos, 15
asistente de modelos, 11
AsNonUnicode, 89
Asociaciones de clave externa, 43
Asociaciones Independientes, 40
AssociationSet, 37
AssociationType, 37

B

BuiltInTypeKind, 109

C

Caché de planes de consulta, 120
Carga perezosa, 415
Cascade, 49
ChangeTracker, 337
ChangeTrackingEnabled, 215
cláusula
  from, 368
  group by, 368
  join, 368
  let, 368
  orderby, 368
  select, 368
  where, 368
Code First, 269
Collection, 342
columna discriminadora, 61
columnas discriminadoras, 56
CommandText, 118
CommandType, 118
CompiledQuery, *412*
ComplexTypeConfiguration, 297, 298
ConcurrencyCheck, 290, 291
ConcurrencyMode, 205, 290
Conformista, 95
consultas parametrizadas, 115
Consultas polimórficas, 116
contenedor de entidades, 125
contrato, 4
convenciones, 277, 278, *279*, 289, 296, 303, 306, 310, 326, 333
Create<TDereviedEntity>,, 344
CSDL, 9

Customer/ Supplier, 95

D

Data Annotations, 282, 288, 348
Data Mapper, 7
Database, 269, *272*, 335, 336, 340, 341
Database Generation Workflow, 83
DataContractResolver, 229
DbContext, *271*, *272*, 273, 274, 275, 279, 280, 283, 285, 295, 300, 324, 332, 333, 334, 335, 336, 337, 343
DbEntityEntry, 337
DbEntityValidationException, 349
DbEntityValidationResult, 349
DbModelBuilder, *279*, *280*, 284, 284, 285, 287, 288, 289, 290, 292, 293, 294, 295, 297, 300, 302, 311, 314, 317, 318, 320, 323, 325, 330, 334, 340, 348
DbPropertyValues, 339
DbSet, *280*, 303, 304, 324, 332, 334, 343, 344, 345, 346
DDD, 1
DDL Generation Template, 83
DefaultConnectionFactory, 272, 275, 336
DeleteObject, 199
desajuste de impedancias, 355
Diseñador de modelos EDM, 9
Domain Driven Design, 1
DropCreateDatabaseAlways, 275, 276, 345
DropCreateDatabaseIfModelChanges, 275
DTO's, 332
duplicación de entidades, 259

E

EDM, 8
EDMGen, 18
edmx, 9
Eliminación de entidades, 199
encuentros
  agrupados, 377
entidad, 2
Entity Client, 5, 19, 98
Entity Data Model Wizard, 10
Entity SQL, 20, 98
EntityCommand, 105, 118
EntityConnection, 105, 125
EntityContainer, 25
EntityFunctions, 88
EntityMappingConfiguration, 325
EntityParameter, 115
EntitySet, 25
EntityState, 185
EntityType, 24, 25
EntityTypeConfiguration, 294
Entry, 337
eSQL, 20, 98, 385
ExecuteDataReader, 105
ExecuteStoreQuery, 129
expansión de consultas, 416
expresiones de consulta
  definición, 357
  sintaxis, 367
expresiones lambda, 361

F

FixupCollection, 142
FK Associations, 43
FROM, 386
función definida en el modelo, 86
Funciones canónicas, 402
Funciones de agregación, 402
Funciones de fecha y hora, 406

Funciones de manipulación de cadenas, 405
Funciones matemáticas, 405

G

GetDatabaseValues, 340
GetValidationResult, 350
GROUP BY, 390

H

Having, 390
herencia, 56
Herencia por Jerarquia, 56

I

IDbConnectionFactory, 272, 273, 274, 275, 336
IExtendedDataRecord, 110
Inicializadores de objetos, 361
INotifyPropertyChanged, 140
Inserción de entidades, 195
IObjectWithChangeTracker, 140
IRelatedEnd, 162
IsConcurrencyToken, 290
IValidatableObject, 352

K

KnownType, 229

L

Lazy Loading, patrón, 164
LazyLoadingEnabled, 415
LIMIT, 115
LINQ, 355, 356
LINQ, operadores de consulta estándar, 369

M

MarkAsAdded, 215
MarkAsDelete, 215
MarkAsUnchanged, 215
materialización, 21, 123
MDF, 86
MergeOption, 188, 418
MetadataItem, 108
MetadataWorkspace, Propiedad, 125
método estático de construcción, 196
métodos de construcción de consultas, 152
métodos extensores, 361, 362
Model Browser, 28
Model Defined Function, 86, 421
Model First, 78
Modelo de Entidades, 8
modelo primero, 78
MSL, 9
multiplicidad, 39

O

Object Services, 21, 123
ObjectContext, 124
ObjectSet<TEntity>, 145
ObjectStateManager, 125
OFTYPE, 117
Operaciones entre bits, 407
Operador de navegación, 401
Operadores aritméticos, 392
Operadores de acceso a miembros, 394
Operadores de comparación, 393
Operadores de construcción de tipos, 397
operadores de consulta, 361
Operadores de consulta estándar, LINQ, 369
Operadores de referencia, 398

Operadores de tipo, 394
Operadores lógicos, 394
OptimisticConcurrencyException, 207
ordenación de consultas, 177
ORDER BY, 391
ORM, 3

P

Parámetros, 391
Patrón Unit Of Work, 124
Plain Old CLR Objects, 141
Plantillas T4, 423
Pluralizar, 27
POCO, Clases, 223
POCO, objetos, 141
Precompilación de vistas, 413
procedimientos almacenados, 118
productos cartesianos, 373
propiedades de navegación, 107
proveedores de datos, 7
ProxyDataContractResolver, 229

Q

Queryable, 381
QueryView, 68

R

RecordAdditionToCollectionProperties, 213
Refactorizar en un tipo complejo, 35
Reference, 342
RefreshMode, 207
Reload, 340
repositorio, 242
Repositorio, 4
Resolución de las llamadas, 364
restricciones referenciales, 45
rowversion, 290
RowVersion, 292

S

SARGABLE, 419
SaveChanges, 196
sección C-S, 9
SELECT, 387
Selft Tracking Entities, 135, 211, 236
Separate-ways, 96
ServiceKnownType, 229
servicios de objetos, 123
servicios del dominio, 4
Shared Kernel, 94
singularizar, 27
SKIP, 115
SQL Builder Methods, 152
SqlCeConnectionFactory, 272, 274
SqlConnectionFactory, 272, 273, 274
SqlFunctions, 88
SSDL, 9
StartTracking, 215
State, 185
StoreGeneratedPattern, 33
StructuralTypeConfiguration, 328
System.Data.Objects, 124
System.Data.Objects.DataClasses, 124

T

Table Per Hierarchy, 57
Table Per Type, 62
Table Splitting, 49
Tipos anónimos, 361
ToObject, 339
ToTraceString, 119
TPC, 324
TPH, 56, 319
TPT, 322
TPT Inheritance, 62

Transaccionalidad, 210
Transacciones, 119, 210
TransactionScope, 119

U

unidad de trabajo, 124
UnintentionalCodeFirstException, 334
Unit Of Work, patrón, 241
UseLegacyPreserveChangesBehavior, 191

V

ventana de mapeo, 28
vista de consulta, 68
vistas, 66

W

WHERE, 389
WriteHeader, 239

Unai Zorrilla Castro
Data Platform Development MVP

Lleva cerca de 10 años desarrollando software como consultor independiente y en diversas empresas privadas, realizando tareas de arquitectura de software bajo plataforma .NET y toca distintas ramas tecnológicas.

Ha sido reconocido con el galardón de Microsoft MVP, colabora activamente con Microsoft en eventos de formación, talleres de arquitectura, guías de producto así como con la publicación de artículos técnicos en revistas especializadas como .NET, MSDN Online, DotNetMania.

Actualmente trabaja como consultor de gran empresa en Plain Concepts, empresa de la cual es socio fundador.

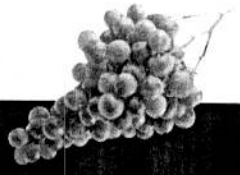
Colaboradores

Yamil Hernández Sitó
Cesar de la Torre Llorente
Pablo Pérez Alter



Todos nuestros autores están galardonados con el título de MVP (Most Valuable Professional) por Microsoft Corporation.

Se trata de un reconocimiento de ámbito mundial que el gigante del software concede cada año a los profesionales más destacados de entre 90 países y 90 tecnologías.



Ya ha pasado mucho tiempo desde la primera versión de ADO.NET Entity Framework, así como de la primera edición de esta obra, la cual supuso el primer libro editado sobre esta tecnología en el mundo. Esta nueva entrega, además de una revisión profunda de los distintos conceptos explicados anteriormente, cubre también las nuevas funcionalidades de EF 4.0 y EF 4.1. Para apoyar a este libro y a los distintos recursos de ejemplos que lo acompañan, se ha incorporado una completa aplicación de referencia en .NET 4.0 en la que se pueden ver muchas de las técnicas y patrones explicados. Esta aplicación -alojada en Codeplex- incorpora también elementos adicionales como WCF, WPF, Silverlight, MVC...

La actual edición de este libro se divide en seis capítulos principales más una serie de apéndices para ampliar algunos elementos. El texto comienza con un capítulo de introducción tanto a la tecnología EF 4.x como a diferentes conceptos importantes que serán manejados a lo largo del resto de libro.

Los siguientes capítulos introducen paulatinamente al lector en las distintas posibilidades que la tecnología ofrece para consultar y actualizar los datos de un modelo conceptual. Todo ello se hace de una forma ordenada, tal y como se concibieron los distintos subsistemas que componen Entity Framework, y explicando las ventajas e inconvenientes de cada uno de ellos.

Se desarrolla un ejercicio de aplicación en el mundo real, apoyándose en la mencionada aplicación de ejemplo que el usuario puede descargar y probar en Codeplex.

Incluye todas las novedades de EF 4.1 haciendo especial hincapié en Code First.

Dado que se profundiza conceptualmente en todos los temas, aparte de ser una herramienta fundamental de aprendizaje, este libro es también un manual de referencia que siempre querrá tener a mano.

Nivel: Intermedio - Avanzado

campus MVP
www.campusmvp.com

krasis PRESS
www.krasispress.com

ISBN 978-84-939036-3-3

