Universidad Nacional Autónoma De Nicaragua, León

Facultad de Ciencias y Tecnología

Departamento de Computación

"Ventajas y Desventajas del Lenguaje de Consulta Integrado (LINQ) en comparación con el lenguaje SQL utilizando la tecnología ADO.NET"

Monografía para optar al título de Ingeniero en Telemática.

Autores:

Br. Ricardo Rigoberto Espinoza Solís.
Br. Kenia Verónica Vargas Canales.

Tutor:

MSC. Hermógenes Ruiz.

Marzo de 2012

DEDICATORIA

A Dios:

Dedico este trabajo primeramente a Dios, por ser el motor que me impulsa cada día, por brindarme salud y fuerzas en los momento más difíciles, por darme la oportunidad de culminar esta meta y no permitirrendirmeante los obstáculos.

A mis padres y hermano:

Porque todo lo que he logrado ha sido gracias a su apoyo incondicional, por alentarme a salir adelante y por ayudarme a cumplir las metas que me he propuesto. Por ser una familia excepcional, quienes han estado presentes en todo momento.

Kenia Verónica Vargas Canales

A Dios:

Dedico este trabajo a Dios porque me ha brindado salud y ha sido mi fuente de sabiduría, porque me ha llevado de la mano, dándome fuerzas y esperanza para salir adelante y llegar hasta el final de esta etapa.

A mis padres y hermanos:

Por todo el apoyo que me han brindado, porque han estado a mi lado en los momentos más difíciles, durante el transcurso de mi vida, por impulsarme cada día, ya que sin su ayuda no hubiera sido posible llegar hasta este momento.

A mis abuelos, primos y tíos:

Porque han sido una familia incondicional, con quienes he contado en todo momento, quienes han compartido conmigo mis triunfos y fracasos.

Ricardo Rigoberto Espinoza Solís

AGRADECIMIENTO

A Dios por darme la vida, salud y facilitarme los medios necesarios para llegar a finalizar esta carrera.

A mis padres y hermano por guiarme por el camino correcto, por proveerme su apoyo en todos los aspectos, por ayudarme a enfrentar los diferentes obstáculos. A mis suegros quienes me han acogido como parte de la familia y han estado a mi lado apoyándome durante esta etapa. A mi esposo, quien me ha ayudado a atravesar los momentos difíciles, que ha sido fundamental en el transcurso de la carrera.

Kenia Verónica Vargas Canales

A Dios por darme la vida, salud y sabiduría para lograr mi objetivo, por permitirme compartir este trabajo con toda mi familia.

A mis padres, por la educación que me han brindado y por proveerme lo necesario para concluir esta etapa de mi formación.

Ricardo Rigoberto Espinoza Solís

INDICE

INTRODUCCION

Los sistemas de bases de datos surgieron en respuesta a los primeros métodos de gestión informatizada de los datos comerciales. Aunque las interfaces de usuario ocultan los detalles del acceso a las bases de datos y la mayoría de las personas no sonconscientes de que están interactuando con una base de datos, el acceso a las bases de datos forma actualmente una parte esencial de la vida de casi todas las personas.

Los programadores han escrito aplicaciones de sistemas en respuesta a las necesidades cada empresa. Los programadores tienen la libertad de elegir el lenguaje de programación que utilizaran, de acuerdo a los beneficios que cada uno de ellos les ofrece.

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

LINQ nos ofrece la posibilidad de expresar las operaciones de consulta en el propio lenguaje y no como literales de cadena pertenecientes a otro lenguaje incrustados en el código de aplicación (por ejemplo, sentencias SQL).

En LINQ to SQL, el modelo de datos de una base de datos relacional se asigna a un modelo de objetos expresado en el lenguaje de programación del programador. El proveedor LINQ to SQL convierte a SQL estas consultas integradas en el lenguaje y las envía a la base de datos para su ejecución. Cuando la base de datos devuelve los resultados, LINQ to SQL los vuelve a convertir en objetos expresados en el propio lenguaje de programación utilizado.

ANTECEDENTES

IBM desarrollo la versión original de SQL, originalmente denominado Sequel, como parte del proyecto System R, a principios de 1970. El lenguaje Sequel ha evolucionado desde entonces y su nombre ha pasado a ser SQL (Structured Query Language, lenguaje estructurado de consultas). Hoy en día, numerosos productos son compatibles con el lenguaje SQL y se ha establecido como el lenguaje estándar para las bases de datos relacionales.

Usando SQL se puede hacer mucho más que consultar las bases de datos, es posible además definir la estructura de los datos, modificar los datos de la base de datos y especificar restricciones de seguridad.

En Noviembre del 2007 junto con el lanzamiento de .NET Framework 3.5 se presentó LINQ (Language INtegrated Query) como una de las novedades de esta versión. Muchas de las características de LINQ fueron originalmente probadas con el lenguaje creado por Microsoft Research denominado Cω.

LINQ proporciona de forma nativa la capacidad de realizar consultas al estilo SQL desde el propio lenguaje de forma agnóstica al propio sistema de almacenamiento, es decir que tanto podemos utilizar sobre orígenes o conjuntos de datos relacionales como objetos almacenados en memoria así como un amplio abanico de proveedores.

JUSTIFICACION

Las bases de datos forman una parte esencial de casi todas las empresas actuales, a lo largo de los años se ha venido incrementando la interacción entre el usuario y las bases de datos. De modo que cada día más personas necesitan acceder a la información que estas contienen, requiriendo menor tiempo de respuesta y mejores beneficios en las consultas realizadas.

En los últimos años ha aumentado la tecnología junto con los lenguajes de programación orientados a objetos, es por ello que hemos decidido determinar las ventajas que nos proporciona utilizar un lenguaje de consulta integrado que se puede ejecutar sobre varias fuentes como DataSet, XML, Objetos de memoria, Bases de datos relacionales, por medio de sus respectivos proveedores, LINQ to DataSet, LINQ to XML, LINQ to Objects, LINQ to SQL y Entity Framework.

Tradicionalmente las consultas con datos se expresan como cadenas sencillas, para lo cual es necesario conocer lenguaje de consultas diferente para cada tipo de origen de datos, como bases de datos, documentos XML, etc. LinQ llega con la finalidad de ofrecer la posibilidad de realizar las consultas en el propio lenguaje y no como otro lenguaje incrustado en el código, lo que facilita consultar y actualizar los datos.

La idea de contar con las capacidades de un lenguaje de consulta integrado para poder manipular los datos en memoria de una forma más ágil presume una gran mejora en el rendimiento de las aplicaciones. Nadie puede discutir la flexibilidad y potencia de las consultas SQL en cuanto a base de datos, pero llevar este concepto a las aplicaciones mejorará en gran manera la programación del lado del cliente.

Mediante este trabajo se pretende evaluar los beneficios, facilidades y tiempos de respuesta que nos ofrece LINQ, para que los programadores tengan la opción de elegir entre un lenguaje capaz de expresar de manera uniforme consultas sobre colecciones de datos de diversa procedencia y la manera usual en que se ha venido accediendo a los datos.

OBJETIVOS

Objetivo general

- Determinar ventajas y desventajas del Lenguaje de Consultas Integrado (LINQ) en comparación con el lenguaje SQL utilizando la tecnología ADO.NET

Objetivos específicos:

- Investigar propiedades y características del lenguaje LINQ.
- Realizar pruebas de desempeño utilizando LINQ y SQL.
- Establecer ventajas y desventajas de LINQ en comparación con el lenguaje SQL.

MARCO TEORICO

BASES DE DATOS

Una base de datos es un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso, además se caracteriza por contener información relevante para una empresa.

Existen programas denominados Sistemas Gestores de Bases de Datos, abreviado SGBD, cuyo objetivo principal es almacenar y posteriormente acceder a los datos de forma rápida y estructurada.

Tipos de Bases de Datos

Las bases de datos pueden clasificarse de varias maneras, de acuerdo al contexto que se esté manejando, la utilidad de las mismas o las necesidades que satisfagan.

***Bases de datos estáticas*:**Son bases de datos de sólo lectura, utilizadas primordialmente para almacenar datos históricos que posteriormente se pueden utilizar.

Bases de datos dinámicas:Éstas son bases de datos donde la información almacenada se modifica con el tiempo, permitiendo operaciones como actualización, borrado y adición de datos, además de las operaciones fundamentales de consulta.

Modelos de Bases de Datos

Las bases de datos además de la clasificación por la función éstas también se pueden clasificar de acuerdo a su modelo de administración de datos.

Un modelo de datos es básicamente una "descripción" de algo conocido como contenedor de datos (donde se guarda la información), así como de los métodos para almacenar y recuperar información de esos contenedores. Los modelos de datos no son cosas físicas: son abstracciones que permiten la implementación de un sistema eficiente de base de datos; por lo general se refieren a algoritmos, y conceptos matemáticos.

Algunos modelos de bases de datos utilizados con frecuencia:

- **Bases de Datos Jerárquicas**
  En este modelo los datos se organizan en una forma similar a un árbol, donde un nodo padre de información puede tener varios hijos. El nodo que no tiene padres es llamado raíz, y a los nodos que no tienen hijos se los conoce como hojas.

  Las bases de datos jerárquicas son especialmente útiles en el caso de aplicaciones que manejan un gran volumen de información y datos muy compartidos, permitiendo crear estructuras estables y de gran rendimiento.

Una de las principales limitaciones de este modelo es su incapacidad de representar eficientemente la redundancia de datos.

- **Bases de Datos de Red**
  Éste es un modelo ligeramente distinto del jerárquico; su diferencia fundamental es la modificación del concepto de nodo, ya que se permite que un mismo nodo tenga varios padres (posibilidad no permitida en el modelo jerárquico).

  Fue una gran mejora con respecto al modelo jerárquico, ya que ofrecía una solución eficiente al problema de redundancia de datos; pero, aun así, la dificultad que significa administrar la información en una base de datos de red ha significado que sea un modelo utilizado en su mayoría por programadores más que por usuarios finales.

- **Bases de Datos Transaccionales**
  Son bases de datos cuyo único fin es el envío y recepción de datos a grandes velocidades, estas bases de datos son poco comunes y están dirigidas por lo general al entorno de análisis de calidad, datos de producción e industrial, es importante entender que su único fin es recolectar los datos a la mayor velocidad posible, por lo tanto la redundancia y duplicación de información no es un problema como con las demás bases de datos, por lo general para poderlas aprovechar al máximo permiten algún tipo de conectividad a bases de datos relacionales.

  Un ejemplo habitual de transacción es el traspaso de una cantidad de dinero entre cuentas bancarias. Normalmente se realiza mediante dos operaciones distintas, una en la que se decrementa el saldo de la cuenta origen y otra en la que incrementa el saldo de la cuenta destino. Para garantizar la atomicidad del sistema, las dos operaciones deben ser atómicas, es decir, el sistema debe garantizar que, bajo cualquier circunstancia (incluso una caída del sistema), el resultado final es que, o bien se han realizado las dos operaciones, o bien no se ha realizado ninguna.

- **Bases de Datos Relacionales**
  Éste es el modelo utilizado en la actualidad para modelar problemas reales y administrar los datos dinámicamente. Su idea fundamental es el uso de "relaciones". Estas relaciones podrían considerarse en forma lógica como conjuntos de datos llamados "tuplas". Pese a que ésta es la teoría de las bases de datos relacionales, la mayoría de las veces se conceptualiza de una manera más fácil de imaginar. Esto es pensando en cada relación como si fuese una tabla que está compuesta por registros (las filas de una tabla), que representarían las tuplas, y campos (las columnas de una tabla).

  En este modelo, el lugar y la forma en que se almacenen los datos no tienen relevancia (a diferencia de otros modelos como el jerárquico y el de red). Esto tiene la considerable ventaja de que es más fácil de entender y de utilizar para un usuario esporádico de la base de datos. La información puede ser recuperada o almacenada mediante "consultas" que ofrecen una amplia flexibilidad y poder para administrar la información.

El lenguaje más habitual para construir las consultas a bases de datos relacionales es SQL, Structured Query Language o Lenguaje Estructurado de Consultas, un estándar implementado por los principales motores o sistemas de gestión de bases de datos relacionales.

Durante su diseño, una base de datos relacional pasa por un proceso al que se le conoce como normalización de una base de datos.

- **Bases de Datos Multidimensionales**
  Son bases de datos ideadas para desarrollar aplicaciones muy concretas, como creación de Cubos OLAP. Básicamente no se diferencian demasiado de las bases de datos relacionales (una tabla en una base de datos relacional podría serlo también en una base de datos multidimensional), la diferencia está más bien a nivel conceptual; en las bases de datos multidimensionales los campos o atributos de una tabla pueden ser de dos tipos, o bien representan dimensiones de la tabla, o bien representan métricas que se desean estudiar.

- **Bases de Datos Orientadas a Objetos**
  Este modelo, bastante reciente, y propio de los modelos informáticos orientados a objetos, trata de almacenar en la base de datos los objetoscompletos (estado y comportamiento).

  Una base de datos orientada a objetos es una base de datos que incorpora todos los conceptos importantes del paradigma de objetos:

  - *Encapsulación* - Propiedad que permite ocultar la información al resto de los objetos, impidiendo así accesos incorrectos o conflictos.

  - *Herencia* - Propiedad a través de la cual los objetos heredan comportamiento dentro de una jerarquía de clases.

  - *Polimorfismo* - Propiedad de una operación mediante la cual puede ser aplicada a distintos tipos de objetos.

  En bases de datos orientadas a objetos, los usuarios pueden definir operaciones sobre los datos como parte de la definición de la base de datos. Una operación (llamada función) se especifica en dos partes. La interfaz (o signatura) de una operación incluye el nombre de la operación y los tipos de datos de sus argumentos (o parámetros). La implementación (o método) de la operación se especifica separadamente y puede modificarse sin afectar la interfaz. Los programas de aplicación de los usuarios pueden operar sobre los datos invocando a dichas operaciones a través de sus nombres y argumentos, sea cual sea la forma en la que se han implementado. Esto podría denominarse independencia entre programas y operaciones.

- **Bases de Datos Documentales**
  Permiten la indexación a texto completo, y en líneas generales realizar búsquedas más potentes. Tesaurus es un sistema de índices optimizado para este tipo de bases de datos.

- **Bases de Datos Deductivas**
  Un sistema de base de datos deductiva, es un sistema de base de datos pero con la diferencia de que permite hacer deducciones a través de inferencias. Se basa principalmente en reglas y hechos que son almacenados en la base de datos. Las bases de datos deductivas son también llamadas bases de datos lógicas, a raíz de que se basa en lógica matemática. Este tipo de base de datos surge debido a las limitaciones de la Base de Datos Relacional de responder a consultas recursivas y de deducir relaciones indirectas de los datos almacenados en la base de datos.

  Este modelo utiliza un subconjunto del lenguaje Prolog llamado Datalog el cual es declarativo y permite al ordenador hacer deducciones para contestar a consultas basándose en los hechos y reglas almacenadas.

  Ventajas
  - Uso de reglas lógicas para expresar las consultas.
  - Permite responder consultas recursivas.
  - Cuenta con negaciones estratificadas.
  - Capacidad de obtener nueva información a través de la ya almacenada en la base de datos mediante inferencia.
  - Uso de algoritmos de optimización de consultas.
  - Soporta objetos y conjuntos complejos.

  Desventajas
  - Crear procedimientos eficaces de deducción para evitar caer en bucles infinitos.
  - Encontrar criterios que decidan la utilización de una ley como regla de deducción.
  - Replantear las convenciones habituales de la base de datos.

  Fases
  - *Fase de Interrogación:* se encarga de buscar en la base de datos información deducible implícita. Las reglas de esta fase se denominan reglas de derivación.

  - *Fase de Modificación*: se encarga de añadir a la base de datos nueva información deducible. Las reglas de esta fase se denominan reglas de generación.

  Interpretación
  Encontramos dos teorías de interpretación de las bases de datos deductivas:

  - *Teoría de Demostración*: consideramos las reglas y los hechos como axiomas.Los hechos son axiomas base que se consideran como verdaderos y no contienen variables. Las reglas son axiomas deductivos ya que se utilizan para deducir nuevos hechos.

  - *Teoría de Modelos:* una interpretación es llamada modelo cuando para un conjunto específico de reglas, éstas se cumplen siempre para esa interpretación. Consiste en asignar a un predicado todas las combinaciones de valores y argumentos de un dominio de valores

constantes dado. A continuación se debe verificar si ese predicado es verdadero o falso.

Mecanismos
Existen dos mecanismos de inferencia:

- *Ascendente:* donde se parte de los hechos y se obtiene nuevos aplicando reglas de inferencia.

- *Descendente:* donde se parte del predicado (objetivo de la consulta realizada) e intenta encontrar similitudes entre las variables que nos lleven a hechos correctos almacenados en la base de datos.

Gestión de bases de datos distribuida (SGBD)

La base de datos y el software SGBD pueden estar distribuidos en múltiples sitios conectados por una red. Hay de dos tipos:

1. ***Distribuidos homogéneos:*** utilizan el mismo SGBD en múltiples sitios.

2. ***Distribuidos heterogéneos:*** Da lugar a los SGBD federados o sistemas multi - bases de datos en los que los SGBD participantes tienen cierto grado de autonomía local y tienen acceso a varias bases de datos autónomas preexistentes almacenados en los SGBD, muchos de estos emplean una arquitectura cliente-servidor.
Estas surgen debido a la existencia física de organismos descentralizados. Esto les da la capacidad de unir las bases de datos de cada localidad y acceder así a distintas universidades, sucursales de tiendas, etc.

LENGUAJE SQL

El lenguaje de consulta estructurado o SQL (*structured query language*) es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones en estas. Una de sus características es el manejo del álgebra y el cálculo relacional permitiendo efectuar consultas con el fin de recuperar de forma sencilla información de interés de una base de datos, así como también hacer cambios sobre ella.

SQL es un lenguaje de acceso a bases de datos que explota la flexibilidad y potencia de los sistemas relacionales permitiendo gran variedad de operaciones.

El lenguaje SQL tiene varios componentes:

- Lenguaje de definición de datos (LDD). Proporciona comandos para la definición de esquema de relación, borrado de relaciones y modificaciones de los esquemas de relación. Existen cuatro operaciones básicas: CREATE, ALTER, DROP y TRUNCATE.
- Lenguaje de manipulación de datos (LMD). Es un lenguaje proporcionado por el sistema de gestión de base de datos que permite a los usuarios llevar a cabo las tareas de consulta o manipulación de los datos, contiene comandos para insertar, borrar y modificar tuplas.
- Integridad. El lenguaje de definición de datos (LDD) incluye comandos para especificar las restricciones de integridad que deben cumplir los datos almacenados en la base de datos. Las actualizaciones que violan las restricciones de integridad se rechazan.
- Definición de vistas. El LDD de SQL incluye comandos para la definición de vistas.
- Control de transacciones. SQL incluye comandos para especificar el comienzo y el final de las transacciones.
- SQL incorporado y SQL dinámico. SQL incorporado y SQL dinámico definen como se pueden incorporar instrucciones de SQL en lenguajes de programación de propósito general como C, C++, C#, Java, etc.
- Autorización. El LDD de SQL incluye comandos para especificar los derechos de acceso a las relaciones y a las vistas.

Estructura básica de las consultas SQL

Las bases de datos relacionales están formadas por un conjunto de relaciones, a cada una de las cuales se le asigna un nombre único. La estructura básica de una expresión SQL consta de tres clausulas: select, from y where.

- La cláusula select se corresponde con la operación de proyección del algebra relacional. Se usa para obtener una relación de los atributos deseados en el resultado de una consulta.
- La cláusula from se corresponde con la operación producto cartesiano del algebra relacional. Genera una lista de las relaciones que deben ser analizadas en la evaluación de la expresión.

- La clausula where se corresponde con el predicado selección del algebra relacional. Es un predicado que engloba a los atributos de las relaciones que aparecen en la clausula from.

Las consultas habituales de SQL tiene la forma

select A_1, A_2,….,A_n
from r_1, r_2,…….r_m
where P

Operaciones sobre conjuntos.

Las operaciones de SQL unión, intersect y except operan sobre relaciones y se corresponden con las operaciones del algebra relaciones. Las relaciones que participan en las operaciones han de ser compatibles, es decir, deben tener el mismo conjunto de atributos.

Funciones de agregación

Las funciones de agregaciones son funciones que toman una colección de valores como entrada y devuelven un solo valor. SQL ofrece cinco funciones de agregación incorporadas:

- Media: avg
- Mínimo: min
- Máximo: max
- Total: sum
- Recuento: count

Los datos de entrada para sum y avg deben ser una colección de números, pero los otros operadores también pueden operar sobre colecciones de datos de tipo no numérico como las cadenas de caracteres.

Valores nulos

SQL permite el uso de valores nulos para indicar la ausencia de información sobre el valor de un atributo. Se puede utilizar la palabra clave especial null en un predicado para comprobar si un valor es nulo.

Subconsultas anidadas.

SQL proporciona un mecanismo para anidar consultas. Las subconsultas son expresiones select-from-where que están anidadas dentro de otra consulta. Una finalidad habitual de las subconsultas es llevar a cabo comprobaciones de pertenencia a conjuntos, hacer comparaciones de conjunto y determinar cardinalidades de conjuntos.

Modificación de la base de datos.

- Borrado. Las solicitudes de borrado se expresan casi igual que las consultas. Solo se pueden borrar tuplas completas; no se pueden borrar solo valores de atributos concretos.

- Inserción. Para insertar datos en una relación, se especifica la tupla que se desea insertar o se formula una consulta cuyo resultado sea el conjunto de tuplas que se desea insertar.Los valores de los atributos de las tuplas que se inserten deben pertenecer al dominio de los atributos.

- Actualizaciones. En determinadas situaciones puede ser deseable modificar un valor dentro de una tupla sin cambiar todos los valores de la misma. Para este tipo de situaciones se puede utilizar la instrucción update. Al igual que insert y delete se pueden elegir las tuplas que se van a actualizar mediante una consulta.

ADO.NET

El ADO.NET es un conjunto de componentes del software que pueden ser usados por los programadores para acceder a datos y a servicios de datos. Es una parte de la biblioteca de clases base que están incluidas en el Microsoft .NET Framework. Es comúnmente usado por los programadores para acceder y para modificar los datos almacenados en un Sistema Gestor de Bases de Datos Relacionales, aunque también puede ser usado para acceder a datos en fuentes no relacionales. ADO.NET es a veces considerado como una evolución de la tecnología ActiveX Data Objects (ADO), pero fue cambiado tan extensivamente que puede ser concebido como un producto enteramente nuevo.

ADO.NET consiste en dos partes primarias:

Data provider

Estas clases proporcionan el acceso a una fuente de datos, como Microsoft SQL Server y Oracle. Cada fuente de datos tiene su propio conjunto de objetos del proveedor, pero cada uno tiene un conjunto común de clases de utilidad:

- Connection: Proporciona una conexión usada para comunicarse con la fuente de datos. También actúa como Abstract Factory para los objetos command.
- Command: Usado para realizar alguna acción en la fuente de datos, como lectura, actualización, o borrado de datos relacionales.
- Parameter: Describe un simple parámetro para un command. Un ejemplo común es un parámetro para ser usado en un procedimiento almacenado.
- DataAdapter: "Puente" utilizado para transferir data entre una fuente de datos y un objeto DataSet.
- DataReader: Es una clase usada para procesar eficientemente una lista grande de resultados, un registro a la vez.

DataSets

Los objetos DataSets, un grupo de clases que describen una simple base de datos relacional en memoria. Las clases forman una jerarquía de contención:

- Un objeto DataSet representa un esquema (o una base de datos entera o un subconjunto de una). Puede contener las tablas y las relaciones entre esas tablas.

- Un objeto DataTable representa una sola tabla en la base de datos. Tiene un nombre, filas, y columnas.

- Un objeto DataView "se sienta sobre" un DataTable y ordena los datos (como una cláusula "order by" de SQL) y, si se activa un filtro, filtra los registros (como una cláusula "where" del SQL). Para facilitar estas operaciones se usa un índice en memoria. Todas las DataTables tienen un filtro por defecto, mientras que pueden ser definidos cualquier número de DataViews adicionales, reduciendo la interacción con la base de datos subyacente y mejorando así el desempeño.

- Un DataColumn representa una columna de la tabla, incluyendo su nombre y tipo.

- Un objeto DataRow representa una sola fila en la tabla, y permite leer y actualizar los valores en esa fila, así como la recuperación de cualquier fila que esté relacionada con ella a través de una relación de clave primaria - clave externa.

- Un DataRowView representa una sola fila de un DataView, la diferencia entre un DataRow y el DataRowView es importante cuando se está interactuando sobre un resultset.

- Un DataRelation es una relación entre las tablas, tales como una relación de clave primaria - clave ajena. Esto es útil para permitir la funcionalidad del DataRow de recuperar filas relacionadas.

- Un Constraint describe una propiedad de la base de datos que se debe cumplir, como que los valores en una columna de clave primaria deben ser únicos. A medida que los datos son modificados cualquier violación que se presente causará excepciones.

Un DataSet es llenado desde una base de datos por un DataAdapter cuyas propiedades Connection y Command que han sido iniciados. Sin embargo, un DataSet puede guardar su contenido a XML (opcionalmente con un esquema XSD), o llenarse a sí mismo desde un XML, haciendo esto excepcionalmente útil para los servicios web, computación distribuida, y aplicaciones ocasionalmente conectadas.

ADO.NET Y Visual Studio.NET

En el IDE Visual Studio .NET existe la funcionalidad para crear las subclases especializadas de las clases del DataSet para un esquema particular de base de datos, permitiendo el acceso conveniente a cada campo a través de propiedades fuertemente tipadas. Esto ayuda a capturar más errores de programación en tiempo de compilación y hace más útil la característica Intellisense del IDE.

PROGRAMACION ORIENTADA A OBJETOS

La programación orientada a objetos o POO es un paradigma de programación que usa objetos y sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, abstracción,polimorfismo y encapsulamiento. Su uso se popularizó a principios de la década de los años 1990. En la actualidad, existe variedad de lenguajes de programación que soportan la orientación a objetos.

Conceptos fundamentales

La programación orientada a objetos es una forma de programar que trata de encontrar una solución a estos problemas. Introduce nuevos conceptos, que superan y amplían conceptos antiguos ya conocidos. Entre ellos destacan los siguientes:

- Clase: definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas.

- Herencia: (por ejemplo, herencia de la clase C a la clase D) Es la facilidad mediante la cual la clase D hereda en ella cada uno de los atributos y operaciones de C, como si esos atributos y operaciones hubiesen sido definidos por la misma D. Por lo tanto, puede usar los mismos métodos y variables públicas declaradas en C. Los componentes registrados como "privados" (private) también se heredan, pero como no pertenecen a la clase, se mantienen escondidos al programador y sólo pueden ser accedidos a través de otros métodos públicos.

- Objeto: entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos) los mismos que consecuentemente reaccionan a eventos. Se corresponde con los objetos reales del mundo que nos rodea, o a objetos internos del sistema (del programa). Es una instancia a una clase.

- Método: Algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un "mensaje". Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un "evento" con un nuevo mensaje para otro objeto del sistema.

- Evento: Es un suceso en el sistema (tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto). El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento, a la reacción que puede desencadenar un objeto, es decir la acción que genera.

- Mensaje: una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.

- Propiedad o atributo: contenedor de un tipo de datos asociados a un objeto (o a una clase de objetos), que hace los datos visibles desde fuera del objeto y esto se define como sus características predeterminadas, y cuyo valor puede ser alterado por la ejecución de algún método.

- Estado interno: es una variable que se declara privada, que puede ser únicamente accedida y alterada por un método del objeto, y que se utiliza para indicar distintas situaciones posibles para el objeto (o clase de objetos). No es visible al programador que maneja una instancia de la clase.

- Componentes de un objeto: atributos, identidad, relaciones y métodos.

- Identificación de un objeto: un objeto se representa por medio de una tabla o entidad que esté compuesta por sus atributos y funciones correspondientes.

En comparación con un lenguaje imperativo, una "variable", no es más que un contenedor interno del atributo del objeto o de un estado interno, así como la "función" es un procedimiento interno del método del objeto.

Características de la Programación Orientada a Objetos

Existe un acuerdo acerca de qué características contempla la "orientación a objetos", las características siguientes son las más importantes:

- Abstracción: denota las características esenciales de un objeto, donde se capturan sus comportamientos.Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar cómo se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos y cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción.El proceso de abstracción permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevos tipos de entidades en el mundo real. La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella podemos llegar a armar un conjunto de clases que permitan modelar la realidad o el problema que se quiere atacar.

- Encapsulamiento: Significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente.

- Modularidad: Se denomina Modularidad a la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes. Estos módulos se pueden compilar por separado, pero tienen conexiones con otros módulos. Al igual que la encapsulación, los lenguajes soportan la Modularidad de diversas formas.

- Principio de ocultación: Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una interfaz a otros objetos que específica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos.

- Polimorfismo: comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre, al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. O dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llamaasignación tardía o asignación dinámica. Algunos lenguajes proporcionan medios más estáticos (en "tiempo de compilación") de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++.

- Herencia: las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo. Esto suele hacerse habitualmente agrupando los objetos enclases y estas en árboles o enrejados que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay herencia múltiple.

- Recolección de basura: la recolección de basura o garbage collector es la técnica por la cual el entorno de objetos se encarga de destruir automáticamente, y por tanto desvincular la memoria asociada, los objetos que hayan quedado sin ninguna referencia a ellos. Esto significa que el programador no debe preocuparse por la asignación o liberación de memoria, ya que el entorno la asignará al crear un nuevo objeto y la liberará cuando nadie lo esté usando. En la mayoría de los lenguajes híbridos que se extendieron para soportar el Paradigma de Programación Orientada a Objetos como C++ u Object Pascal, esta característica no existe y la memoria debe desasignarse manualmente.

LINQ

Language-Integrated Query (LINQ) es una innovación introducida en Visual Studio 2008 y. NET Framework versión 3.5 que sirve de puente entre el mundo de los objetos y el mundo de los datos, se crea con la finalidad de ofrecer la posibilidad de realizar las operaciones de consulta en el propio lenguaje y no como otro lenguaje incrustado en el código.

El objetivo de crear LINQ es permitir que todo el código hecho en Visual Studio (incluidas las llamadas a bases de datos, datasets, XMLs) sea orientado a objetos. Anteriormente la manipulación de datos externos tenía un concepto más estructurado que orientado a objetos, para ello LINQ trata de facilitar y estandarizar el acceso a dichos objetos.

LINQ define operadores de consulta estándar que permiten a lenguajes habilitados con el mismo filtrar, enumerar y crear proyecciones de varios tipos de colecciones usando la misma sintaxis. Tales colecciones pueden incluir arrays, clases enumerables, XML, conjuntos de datos desde bases de datos relacionales y otros orígenes de datos.

En muchas ocasiones es necesario acceder a diferentes dominios como: documentos XML, bases de datos, etc, en la cual cada dominio de datos tiene su propio modelo de acceso, por ejemplo Base de datos – SQL, XML- DOM, XQuery. Además de los diferentes enfoques para representar los datos, como Modelo Relacional, Modelo Jerárquico, entre otros.

RECURSOS DEL LENGUAJE COMPATIBLE CON LINQ

Estas características son la base para formar las expresiones de consulta, las cuales constituyen el principal reflejo en el lenguaje de la tecnología LINQ.

Los elementos básicos sobre los que se construyeron estas nuevas extensiones que se incluyeron a partir de C# 3.0 son los siguientes:

1. ***Declaración implícita de variables locales.***

   A las variables locales se les puede asignar un tipo utilizando el identificador *var*. Dicho identificador le indica al compilador que deduzca el tipo de la variable a partir de la expresión que se encuentra en el lado derecho de la instrucción de inicialización, es decir, es posible declarar una variable inicializada con un valor sin especificar su tipo, la variable tomara el tipo del valor con el que se inicializa. Por ejemplo:

```
// x es tomado como string
var x = "hola";

// y es tomado como int
var y = 5;
```

2. Matrices de tipos definidos de forma implícita.

También, es posible crear una matriz sin especificar explícitamente el tipo de los elementos, el tipo de la matriz se deduce a partir del tipo de los valores de los elementos proporcionados, Por ejemplo:

```
var a = new [ ] {5, 10, 15};  // Tipo int[]
var b = new [ ] {"hola", "mundo"};  // Tipo string[]
```

3. Tipos anónimos.

Los tipos anónimos ofrecen una manera útil de encapsular un conjunto de propiedades de solo lectura en un único objeto, sin tener que definir antes un tipo de forma explícita. El compilador deduce el tipo de cada propiedad.

Un tipo anónimo es una manera cómoda de agrupar temporalmente un conjunto de propiedades en un resultado de una consulta sin tener que definir un tipo con nombre independiente. Se utilizan en la cláusula select de una expresión de consulta para devolver un subconjunto de las propiedades de cada objeto de la secuencia de origen.

Normalmente, cuando se usa un tipo anónimo para inicializar una variable, se declara la variable como variable local tipificada implícitamente utilizando var.

```
var persona = new {Nombre = "María", Edad = 24};
```

4. Propiedades auto – implementadas.

Las propiedades auto-implementadas hacen que la declaración de la misma sea breve y concisa cuando no se requiere ninguna lógica adicional en los descriptores de acceso de la propiedad. Cuando se declara una propiedad auto-implementada, el compilador crea un atributo de respaldo privado y anónimo al que solo se puede acceder mediante los descriptores *get y set*. Por ejemplo:

```
// Propiedades auto implementadas
public string Nombre { get; set;}
public DateTimeFechaNac { get; set; }
```

5. Iniciadores de objetos y colecciones.

Los iniciadores de objeto y de colección permiten asignar valores a los campos de de los objetos al momento de crearlos, sin tener que llamar explícitamente al constructor del objeto. Los iniciadores suelen utilizarse en expresiones de consulta. Por ejemplo:

```
public class CPersona
{
        // Propiedades auto implementadas
        public string Nombre { get; set;}
        public DateTime FechaNac { get; set; }
```

```
}

var  persona = new CPersona {Nombre = "María", FechaNac = new
DateTime (1987, 12, 10)};
```

En el ejemplo anterior se observa que no es necesario llamar al constructor del objeto, sino que es posible inicializar el objeto asignándole los valores correspondientes.

También es posible inicializar una colección de la siguiente manera:

```
List<CPersona> listPersona = new List<CPersona> {
        new CPersona  {
        Nombre = "Kenia", FechaNac = new DateTime (1987, 12, 10)},
        new CPersona  {
        Nombre = "Ricardo", FechaNac = new DateTime (1987, 9, 22)}};
```

En el ejemplo anterior se inicializa la colección listPersona inicializando cada uno de los objetos de la colección.

6. Métodos extensores

Los métodos extensores nos permiten extender o agrega más funcionalidad a una clase existente, sin necesidad de recurrir a la herencia (técnica que en ocasiones es imposible de utilizar, por ejemplo, si la clase ha sido calificada como sealed). Para crear un método extensor debemos utilizar el modificador this como primer parámetro del método, ya que es this quien indica que se trata de un método extensor. Además, la clase que contiene los métodos extensores y el método, deben ser declarados como static.

```
public static class CPersonaEx
{
       public static int Edad (this CPersona persona)
       {
          int edad = DateTime.Today.Year - persona.FechaNac.Year;

          if (DateTime.Today.Month > persona.FechaNac.Month ||
          DateTime.Today.Month == persona.FechaNac.Month &&
          DateTime.Today.Day >= persona.FechaNac.Day)
   return edad;
   return edad -1;
          }
}
```

En el siguiente ejemplo un elemento de una lista utiliza el método extensor anterior y se observa que el comportamiento del método extensor es como si fuera un método nativo de CPersona.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
    public static class CPersonaEx
    {
            public static int Edad (this CPersona persona)
            {
               int edad = DateTime.Today.Year - persona.FechaNac.Year;

               if (DateTime.Today.Month > persona.FechaNac.Month ||
               DateTime.Today.Month == persona.FechaNac.Month &&
               DateTime.Today.Day >= persona.FechaNac.Day)
        return edad;
        return edad -1;
               }
        }


public sealed class CPersona
{
       // Propiedades auto implementadas
       public string Nombre { get; set; }
       public DateTime FechaNac { get; set; }
}

class Prueba
{
        public static void Main(string[] args)
       {
        List<CPersona> listPersona = new List<CPersona> {
              new CPersona  {
                 Nombre = "Kenia", FechaNac = new DateTime (1987, 12, 10)},
              new CPersona  {
                 Nombre = "Ricardo", FechaNac = new DateTime (1987, 9, 22)}};

       Console.WriteLine(listPersona[0].Edad());
        }
    }
```

Los métodos extensores en la resolución de llamadas por parte del compilador, tiene menor resolución que los métodos nativos; esto es, si la clase CPersona tuviera un método nativo Edad, se utilizaría este.

7. Expresiones lambda

Las expresiones lambda, un recurso tradicional en los lenguajes de programación funcional, es una función anónima que puede contener expresiones e instrucciones, las que se pueden utilizar para crear delegados o tipos de arboles de expresión. Las expresiones lambda se utilizan principalmente con las expresiones de consulta LINQ.

Cuando se define una expresión lambda se utiliza el operador lambda => que se lee como "*va a*", donde en la parte izquierda del operador estarán indicadas las variables o parámetros de la función anónima y a la derecha se indicara el código de la función, es decir la expresión o bloque de instrucciones. A continuación se muestra la sintaxis de esta expresión:

(Lista de parámetros) => Expresión o Bloque de instrucciones

El operador lambda => tiene la misma prioridad que la asignación (=) y es asociativo por la derecha.Pueden reemplazar de una manera más concisa a los métodos anónimos: bloques de código que pueden colocarse inline en aquellos sitios donde el compilador espera encontrarse un delegado.

La lista de parámetros, separados por comas, ira entre paréntesis (estos pueden omitirse cuando se trate de un único parámetro).. A continuación se escribe el símbolo de implicación y después, la expresión o bloque de sentencias a ejecutar, tomando como argumentos esos parámetros. Por ejemplo:

```
A => a * 2                          //expresión; tipo implícito
(int a) => a * 2                    //expresión; tipo explicito
A => { return a * 2; }              //sentencia; tipo implícito
(int a) => { return a * 2; }        //sentencia; tipo explicito
( a, b ) => a + b                   //varios parámetros
( ) => Console.WriteLine( )         //sin parámetros
(string s, int x) => s.Length > x   //varios parámetros con tipo
```

Cuando el compilador no pueda deducir los tipos de entrada, habrá que especificarlos explícitamente, como ocurre en el último ejemplo.

El cuerpo de una lambda de sentencias puede estar compuesto por cualquier número de sentencias, pero generalmente este número es muy pequeño.

```
public delegate void Suma(int a, int b);

//…
Suma deSuma = (x, y) => { var s = x + y; Console.WriteLine(s); };
 deSuma(5, 6);//imprime 11
```

Como ejemplo, vamos a añadir a la clase CPersona un método NacidoAño que muestre las personas de una lista que haya nacido en un año determinado. El primer argumento de este método será una referencia a la lista de personas y el segundo un delegado que ejecutará la expresión condicional necesaria.

```
//Delegado
public delegate bool ExprCondicional (CPersona pers);
public sealed class CPersona
{
    public string Nombre { get; set; }
    public DateTime FechaNac { get; set; }

     public static void NacidosAño(List<CPersona> lista, ExprCondicional
    condicion)
    {
           foreach(CPersona p in lista)
           {
                  if (condicion (p))
                         Console.WriteLine(p.Nombre);
           }
    }
}
```

```
class Test
{
       public static void Main (string[ ]  args)
       {
        List<CPersona> listPersona = new List<CPersona> {
                 new CPersona  {
                      Nombre = "Kenia", FechaNac = new DateTime (1987, 12, 10)},
              new CPersona  {
                      Nombre = "Ricardo", FechaNac = new DateTime (1987, 9, 22)}};

          CPersona.NacidosAño(listPersona, delegate (CPersona p)
{ return p.FechaNac.Month > 9;});
       }
}
```

La llamada a NacidosAño desde Test, incluye en su segundo argumento un método anónimo. Reemplacemos este método por una expresión lamdba:

```
CPersona.NacidosAño(listPersona, (CPersona p) => p.FechaNac.Month > 9);
```

O bien,

```
CPersona.NacidosAño(listPersona, (p) => p.FechaNac.Month > 9);
```

El tipo de p se deduce de la definición del delegado. A modo de resumen, podemos deducir que las expresiones lambda comparadas con los métodos anónimos tienen las siguientes ventajas: notación mas concisa, expresiva y funcional; permiten definir los parámetros de forma implícita; el cuerpo puede ser una expresión o un bloque de sentencias; y pueden almacenarse en memoria como arboles de expresiones, concepto que estudiaremos un poco más adelante.

El delegado Func<t, TResu>

Este delegado puede usarse para representar un método que puede pasarse como parámetro sin declarar explícitamente ningún delegado personalizado. El método encapsulado debe corresponder a la firma del método definida por el delegado. Esto significa que el método encapsulado debe tener un parámetro, el cual se le pasa por valor, y debe devolver un valor.

El delegado Func utiliza parámetros de tipo:

Public delegate TResult Func<T, …, TResult> (T arg, …);

El valor devuelto siempre se corresponde con el parámetro especificado en el último lugar; el resto son parámetros de entrada. Por ejemplo el siguiente código define un delegado Func que cuando se invoca, devuelve verdadero o falso para indicar si el parámetro de entrada es un número par.

```
Func<int, bool> esPar = n => n % 2 == 0;
Bool resultado = esPar(11); // retorna false
```

Al usar el delegado Func(Of T, TResult), no es necesario definir explícitamente un delegado que encapsule un método con un solo parámetro. En el siguiente ejemplo

podemos observar que al no utilizar el delegado Func se tiene que definir explícitamente un delegado.

```
using System;

public delegate int deSuma(int x, int y);
public class Test
{
    public static void Main()
   {
   // Instancia del delegado para referenciar al método Suma
    deSuma instSuma = Suma;
    int x = 5;
    int y = 6;
   // Se utiliza la instancia del delegado para llamar al método Suma
   Console.WriteLine(instSuma(x , y));
   }

   private static int Suma(int x, int y)
   {
   return x + y;
   }
}
```

El siguiente ejemplo crea una instancia de un delegado Func<t, TResu> que hace referencia al método Suma, no es necesario definir un delegado explícitamente.

```
using System;

public class Test
{
   public static void Main()
   {
   // Instancia del delegado para referenciar al método Suma
   Func<int, int, int> instSuma = Suma;
   int x = 5;
   int y = 6;
   // Se utiliza la instancia del delegado para llamar al método Suma
   Console.WriteLine(Suma(x,y));
   }

   private static int Suma(int x, int y)
   {
   return x + y;
   }
}
```

Utilizando expresiones lambda en el ejemplo anterior:

```
using System;

public class Test
{
   public static void Main()
   {
   // Instancia del delegado Func utilizando una expresión lambda
   Func<int, int, int> instSuma = (a , b) => a + b;
   int x = 5;
   int y = 6;
   // Se utiliza la instancia del delegado Func para calcular la suma
```

```
    Console.WriteLine(instSuma(x , y));
    }

}
```

En la clase CPersona podemos modificar el método NacidosAño para que utilice un delegado Func:

```
public static void NacidosAño(List<CPersona> lista, Func<CPersona, bool>
condicion)
{
    foreach (CPersona p in lista)
    {
           if (condicion(p))
                  Console.WriteLine(p.Nombre);
    }
}
```

La llamada a este método seria:

CPersona.NacidosAño(listPersona, (p) => p.FechaNac.Month> 1987);

Podemos prescindir del método NacidosAño utilizando un delegado Func con dos parámetros de entrada y el tercero de salida.

```
    Func<CPersona, int, bool>esNacidoEnAño = (p, año) => p.FechaNac.Month >
    año;
    foreach (CPersona p in listPersona)
    {
           if(esNacidoEnAño (p, 1987))
                  Console.WriteLine(p.Nombre);
    }
```

8. Operadores de consulta

Los operadores de consulta son los métodos que forman el modelo de LINQ. Incluyen operaciones de filtrado, proyección, agregación, ordenación y otras. Algunos de ellos son (se indica entre paréntesis, la palabra clave C#, si existe, utilizada en las expresiones de consulta que veremos más adelante):

- Select (select). Proyecta valores basados en una función de transformación.
- SelectMany (utiliza varias clausulas from). Proyecta secuencias de valores basados en una función de transformación y, a continuación, los condensa en una sola secuencia (un enumerable).
- Where (where). Selecciona los valores basados en una función de predicado.
- Order by (orderby). Ordena los valores de forma ascendente.
- Join (join … in … on … equals …). Combina dos secuencias según las funciones del selector de claves y extrae pares de valores.
- GroupBy (group … by). Agrupa los elementos que comparten un atributo común. Cada grupo se representa mediante un objeto IGrouping<TKey, T>.
- Count. Cuenta los elemento de una colección y, opcionalmente, solo aquellos que satisfacen una función de predicado.
- Max. Determina el valor máximo de una colección.
- Min. Determina el valor mínimo de una colección.

- Sum. Calcula la suma de los valores de una colección.
- TakeWhile. Devuelve los elementos de una colección mientras que el valor de la condición específica sea true.

La mayoría de estos métodos funcionan sobre objetos cuyo tipo implementa la interfaz IEnumerable<T> (interfaz que proporciona iteración simple en una colección de tipo especificado) o la interfaz IQueryable<T> (interfaz para evaluar consultas con respecto a un origen de datos; hereda de IEnumerable<T>). Se definen como métodos extensores del tipo sobre el que operan. Esto significa que pueden ser llamados utilizando la sintaxis del método estático (o de clase) o la sintaxis del método de un objeto (o instancia).

Muchos operadores de consulta tienen un parámetro de entrada de tipo Func. En base a esto, veamos un ejemplo de cómo se utiliza uno de estos operadores de consulta, por ejemplo, Count:

```
int[] números = { 6, 7, 11, 9, 8, 5, 4, 1, 3, 2 };
int numerosPares = números.Count (n => n % 2 == 0 );  // resultado: 4
```

La expresión lambda utilizada cuenta aquellos enteros(n) que divididos por dos dan como resto 0. Observe también que el compilador puede deducir el tipo del parámetro de entrada; también se puede especificar explícitamente como en el ejemplo siguiente.

El método siguiente generara una secuencia que contiene todos los elementos de la matriz de números que aparecen antes del 11 menores que 9 (6 y 7), ya que 11 s el primer número de la secuencia que no cumple la condición:

```
int[] números = { 6, 7, 11, 9, 8, 5, 4, 1, 3, 2 };
var primerosNumsMenoresQue9 = números.TakeWhile((int n) => n < 9);
foreach(var i in primerosNumsMenoresQue9)
     Console.WriteLine(i);
```

Las llamadas a los métodos de consulta se pueden encadenar en una sola consulta, lo que permite hacer consultas bastante complejas. Además, según vimos anteriormente, algunos de los operadores de consulta más frecuentemente utilizados poseen una sintaxis de palabras clave especificas del lenguaje C# que les permite ser llamados como parte de una expresión de consulta. Por ejemplo:

```
int[] números = { 6, 7, 11, 9, 8, 5, 4, 1, 3, 2 };
var numsImparesOrdenados =
     from n in números
     where (n % 2 != 0)
     orderby n
     select n;

 foreach (var i in numsImparesOrdenados)
     Console.WriteLine(i); // resultado: 1, 3, 5, 7, 9, 11
```

Una expresión de consulta constituye una forma diferente de expresar una consulta, más legible que su equivalente basada en métodos. Las clausulas de las expresiones de consulta se traducen en llamadas a los métodos de consulta en tiempo de

compilación. Por ejemplo, la consulta anterior utilizando llamadas a métodos podría realizarse así:

```
int[] números = { 6, 7, 11, 9, 8, 5, 4, 1, 3, 2 };
var numsImparesOrdenados = números.Where(n => n % 2 != 0).OrderBy(n =>
n).Select(n => n);
```

O, de una forma más cercana a la expresión de consulta anterior, así:

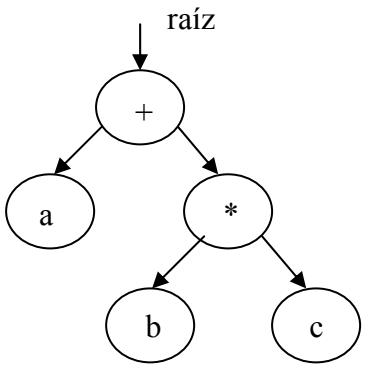
```
var numsImparesOrdenados = números.
       Where(n => n % 2 != 0).
       OrderBy(n => n).
       Select( n => n );
```

En esta otra versión observamos claramente, tipos anónimos, expresiones lamdba, etc. Como base de la expresión de consulta, cuestión que ya indicamos anteriormente.

9. ***Arboles de expresiones lambda***

Los árboles de expresión lambda representan el código en una estructura de datos similar a un árbol, donde cada nodo es una expresión; por ejemplo, una llamada a un método o una operación binaria como x < y. El código representado en árboles de expresión se puede compilar y ejecutar. Esto permite realizar cambios dinámicos en código ejecutable, ejecutar consultas LINQ en varias bases de datos y crear consultas dinámicas.

Los arboles de expresiones lambda permiten representar expresiones lambda como estructuras de datos en lugar de como código ejecutable. Esto es, las expresiones lambda pueden transformarse en arboles de expresiones para su posterior manipulación, almacenamiento o transmisión. Por ejemplo, la expresión a + b * c podemos verla bajo la siguiente estructura en la que los datos se almacenan en forma de árbol (el lector puede reconstruir la expresión recorriendo el árbol de la figura en *in-orden*):



Cuando una expresión lambda esta asignada a una variable de tipo Expression<TDelegado>, por ejemplo Expression<Func>, el compilador emite un árbol de expresión que representa dicha expresión lambda. Por ejemplo algunos operadores de consulta que se definen en la clase Queryable tienen parámetros de tipo Expression<TDelegado>. Entonces, cuando se llama a estos métodos, la lambda pasada como argumento se compilara produciendo un árbol de expresión.

En LINQ, los árboles de expresiones lambda se utilizan para representar consultas estructuradas para orígenes de datos que implementan IQueryable<T>. Por ejemplo, el proveedor LINQ to SQL implementa esta interfaz para realizar consultas en almacenes de datos relacionales. Los compiladores de Visual Basic y C# compilan las consultas destinadas a esos orígenes de datos en código que genera un árbol de expresión en tiempo de ejecución. El proveedor de la consulta puede entonces recorrer la estructura de datos del árbol de expresión y traducirla a un lenguaje de consulta apropiado para el origen de datos.

Solo se pueden ejecutar los arboles de expresiones que representen expresiones lambda, los cuales son de tipo Expression<TDelegado> (por ejemplo, Expression<Func<CPersona, bool>>) o LambdaExpression. Para ejecutar un árbol de expresión primero hay que compilarlo invocando a su método Compile con el fin de crear un delegado ejecutable y después hay que invocar al delegado para que se ejecute.

El ejemplo siguiente asigna una expresión lambda a la variable *exprNacidoEnAño* de tipo Expression<Func<CPersona, bool>> a partir de la cual el compilador generara un árbol de expresión lambda con la finalidad de obtener de una lista de personas las que han nacido en un año determinado.

```
String personas1991 = “ ”;
Expression<Func<CPersona, bool>> exprNacidoEnAño = (p) => p.FechaNac.Year
== 1991;
Foreach (CPersona p in listPersona)
{
       Bool esNacidoEn = exprNacidoEnAño.Compile().Invoke(p);
       If (exprNacidoEnAño.Compile().Invoke(p))
       Personas 1991 += p.Nombre + Environment.NewLine;
}
```

Los arboles de expresiones son útiles para crear consultas dinámicas de LINQ necesarias cuando no se conocen los detalles de la consulta durante la compilación; por ejemplo, porque sea necesario especificar durante la ejecución uno o más predicados para filtrar los datos que se desean obtener, lo que implica crear la consulta durante la ejecución. Pues bien, para crear los arboles de expresiones tendremos que utilizar la funcionalidad proporcionada por las clases del espacio de nombres System.Linq.Expressions.

La clase Expression proporciona la funcionalidad necesaria para crear nodos del árbol de expresión de tipos específicos; por ejemplo, un objeto ParameterExpression, que representa una expresión de parámetro con nombre, un objeto MemberExpression, que representa un acceso a un campo o propiedad, un objeto ConstantExpression, que representa una expresión que tiene un valor contante, un objeto MethodCallExpression, que representa una llamada a un método, un objeto ConditionalExpression, que representa una expresión que tiene un operador condicional, un objeto BinaryExpression, que representa una expresión que tiene un operador binario, o un objeto LambdaExpression, que describe una expresión lambda; todos definidos en el espacio de nombres System.Linq.Expressions. Estas clases se derivan de la clase abstracta Expression.

EXPRESIONES DE CONSULTA

LINQ es una combinación de extensiones al lenguaje y bibliotecas de código administrativo que permiten expresar de manera uniforme "consultas" sobre colecciones de datos de diversa procedencia (objetos en memoria, bases de datos relacionales o documentos XML) utilizando recursos del propio lenguaje de programación. Estas consultas son llevadas a cabo mediante lo que se denomina *expresiones de consulta*. Una consulta es una expresión que recupera datos de un origen de datos.

Una expresión de consulta es una consulta expresada en sintaxis de consultas. Una expresión de consulta está compuesta de un conjunto de cláusulas escritas en una sintaxis declarativa similar a SQL o XQuery. Cada cláusula contiene a su vez una o más expresiones de C#, y estas expresiones pueden ser una expresión de consulta o contener una.

LINQ facilita a los programadores un modelo coherente para trabajar con los datos de varios tipos de formatos y orígenes de datos. En una consulta LINQ, siempre se trabaja con objetos. Se utilizan los mismos modelos de codificación básicos para consultar y transformar datos de documentos XML, bases de datos SQL, conjuntos de datos ADO.NET, colecciones .NET y cualquier otro formato para el que haya disponible un proveedor LINQ.

Las expresiones de consulta responden a una nueva sintaxis que se ha añadido al lenguaje C# y que pueden actuar sobre objetos que implementen la interfaz IEnumerable<T> o IQueryable<T>, entre los que se incluyen las matrices, transformándolos mediante un conjunto de operaciones en otras colecciones que implementen la misma interfaz.

Para explicar las expresiones de consulta, se han definido las clases CPais y CPersona así:

```
Public sealed class CPais
{
       Public string Codigo { get; set; }
       Public string Nombre { get; set; }

}

Public sealed class CPersona
{
       Public string Nombre { get; set; }
       Public DateTime FechaNac { get; set; }
       Public string PaisNac { get; set; }
}
```

Desde el punto de vista de LINQ una consulta no es más que una expresión que recupera datos de un origen de datos determinado. Una cosa es una consulta y otra distinta los resultados que genera. Todas las operaciones de consulta LINQ se componen de tres acciones distintas:

1) **Obtención del origen de datos**. Por ejemplo:

```
List<CPersona> listPersona = new List<CPersona> {
       new CPersona  {
              Nombre = "Kenia", FechaNac = new DateTime (1987, 12, 10)},
       new CPersona  {
              Nombre = "Ricardo", FechaNac = new DateTime (1987, 9, 22)}};
```

2) **Creación de la consulta**. La consulta especifica qué información se va a recuperar de uno o varios orígenes de datos. Opcionalmente, una consulta también especifica cómo debería ordenarse, agruparse y darse forma a esa información antes de ser devuelta. Una consulta se almacena en una variable de consulta y se inicializa con una expresión de consulta. Por ejemplo:

```
var personas1987 =
       From p in listPersona
       Where p.FechaNac.Year == 1987
       Orderby p.Nombre
       Select new { Nombre = p.Nombre };
```

3) **Ejecución de la consulta.**La variable de consulta sólo almacena los comandos de la consulta. Dado que la propia variable de consulta nunca contiene los resultados de la consulta, se puede ejecutar tantas veces como se desee.

   En LINQ, una consulta no se ejecuta hasta que se recorre en iteración la variable de consulta en una instrucción foreach, es decir la ejecución de la consulta es distinta de la propia consulta; Además, la instrucción foreach es donde se recuperan los resultados de la consulta, es decir, no se recuperan los datos con solo crear la variable de consulta (personas1987) sino que hay que ejecutarlaen una instrucción foreach:

```
foreach (var persona in personas1987)
       Console.WriteLine(persona.Nombre);
```

   En el ejemplo siguiente se muestra como se expresan las tres partes de una operación de consulta en una aplicación concreta. En este ejemplo se utiliza una matriz de objetos como origen de datos, sin embargo los mismos conceptos se aplican a otros orígenes de datos. En dicho ejemplo, se realizara una consulta sobre la colección List que se ha venido utilizando en los ejemplos anteriores, con el fin de obtener la colección de personas nacidas en un año determinado.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public sealed class CPais
{
       public string Codigo { get; set; }
       public string Nombre { get; set; }
}

public sealed class CPersona
{
       public string Nombre { get; set; }
       public DateTime FechaNac { get; set; }
```

```
        public string PaisNac { get; set; }
    }
    class Test
    {
    public static void Main (string [ ] args)
    {
        List<CPais> listPais = new List<CPais> {
            new CPais {
                Codigo = "ES", Nombre = "España" },
            new CPais {
                Codigo = "NI", Nombre = "Nicaragua" },
            new CPais {
                Codigo = "FR", Nombre = "Francia" },
            new CPais {
                Codigo = "US", Nombre = "Estados Unidos" },
    };

List<CPersona> listPersona = new List<CPersona> {
        new CPersona  {
            Nombre = "Kenia", FechaNac = new DateTime (1987, 12, 10)},
        new CPersona  {
            Nombre = "Ricardo", FechaNac = new DateTime (1987, 9, 22)}};

    // Expresion de consulta
    var personas1987 =
        from p in listPersona
        where p.FechaNac.Year == 1987
        orderby p.Nombre
        select new { Nombre = p.Nombre };

    foreach (var persona in personas1987)
        Console.WriteLine(persona.Nombre);
    }
    }
```

Observando la expresión de consulta. A cualquiera que esté familiarizado con la sentencia SELECT de SQL, le habrá resultado fácil entender dicha expresión. Quizás nos sorprenda que la clausula select no esté al principio. La razón es que si estuviera al principio sería imposible ofrecer la ayuda inteligente a la hora de escribir la expresión de consulta, porque aun no se habría especificado la colección de objetos sobre los que se ejecutara la consulta. Esto es, si listPersona es de tipo List<CPersona> (List implementa la interfaz IEnumerable) entonces se deduce que p es de tipo CPersona, con lo que el sistema podrá verificar la sintaxis cuando a continuación escribamos el resto de las clausulas.

La consulta del ejemplo anterior devuelve todos los nombres de la lista listPersona (objeto List) de objetos CPersona ordenados ascendentemente. Contiene cuatro clausulas: from, where, orderby, select. La clausula from especifica el origen de datos y una variable local que representa cada elemento en la secuencia origen, la clausula where aplica el filtro (se pueden utilizar los operadores lógicos), la clausula orderby ordena el resultado ateniéndose al dato especificado y la clausula select especifica el tipo de los elementos devueltos. No olvide que definir la variable de consulta no realiza ninguna acción ni devuelve datos, simplemente almacena la información necesaria para generar los resultados cuando la consulta se ejecute posteriormente.

El ejemplo anterior también podrá haberse escrito así:

```
var personas1987 =
       From p in listPersona
       Where p.FechaNac.Year == 1987
       Orderby p.Nombre
       Select p.Nombre;

Foreach (var persona in personas1987)
   Console.WriteLine(persona);
```

¿Cuál es la diferencia? En el ejemplo anterior personas1987 era una colección de objetos de tipo anónimo con una propiedad Nombre de tipo string y en este, personas1987 es una colección de objetos string.

Compilación de una expresión de consulta

Cuando el compilador encuentra una expresión de consulta la transforma en una consulta compuesta por llamadas a métodos. Por ejemplo:

```
var personas1987 = listPersona.
       where (p => p.FechaNac.Year == 1987).
       orderby (p => p.Nombre).
       select (p => new { Nombre = p.Nombre });
```

En el código anterior observamos llamadas a los métodos Where, OrderBy y Select. Observemos la firma de uno de ellos:

```
public static IEnumerable<TOrigen> where <TOrigen>(
       this IEnumerable<TOrigen> origen,
       Func <TOrigen, bool> predicado)
```

Se trata de un método extensor, que por definición, se puede invocar con un método estático (se ha declarado static) o como un método de un objeto (es una extensión a los métodos del tipo IEnumerable). El primer parámetro precedido por this hace referencia al tipo extendido, en este caso al tipo IEnumerable ya que hemos partido de que los objetos sobre los que operan las expresiones de consulta tienen que implementar esta interfaz, y el segundo, el predicado (el filtro), es un delegado que hace referencia a un método que recibirá como argumento un objeto de tipo TOrigen y devolverá un valor del tipo bool. La función lambda es la que se transformará en ese delegado anónimo.

¿Por qué estos métodos se han implementado como métodos extensores? Pues para que LINQ sea una arquitectura abierta y extensible. Según esto, añadir nuestro propio método Where es tan sencillo, como añadir a nuestra aplicación una clase static con un método static con un primer parámetro precedido por el modificador this. Añadiendo a la aplicación anterior la siguiente clase para extender el tipo IEnumerable con el siguiente método Where:

```
public static class CEnumerableEx
{
       public static IEnumerable<T> where<T> (
       this IEnumerable<T> origen,
       Func<T, bool> predicado)
       {
              foreach (T p in origen)
```

```
                if(predicado(p)) {yield return p;}
        }
}
```

Obsérvese la sentencia *yield return*. Esto hace que el resultado de una consulta no se obtenga hasta el momento en que se produzca la iteración sobre ella; por ejemplo, ejecutando una sentencia foreach.

Si compila y ejecuta ahora la aplicación, observara que al ejecutarse la consulta, el método Where invocado es este que hemos añadido. Precisamente en esto consiste la arquitectura abierta de LINQ: cualquiera puede añadir otros operadores de consulta siempre que cumplan con las firmas que exige el compilador. Esta ha sido la vía a través de la cual se han integrado en el lenguaje las expresiones LINQ.

Sintaxis de las expresiones de consulta

Una expresión de consulta siempre comienza con la clausula from … in, esta especifica un origen de datosjunto con una variable local que representa a cada elemento en la secuencia de origen y a continuación se pueden escribir una o más clausulas from … in, let, where, join … in …on … equals, join … in … on … equals … into, orderby …[ascending], select, group .. by, o group … by … into. Opcionalmente, a final de la expresión puede escribirse una clausula de continuación que comienza con into y continua con el cuerpo de otra clausula.

Una expresión de consulta debe finalizar con una cláusula select o una cláusula group.

Clausula group

La cláusula group se utiliza para generar una secuencia de grupos organizada por una clave especificada, es decir, la clausula group permite agrupar los resultados según la clave que se especifique. Por ejemplo, la siguiente expresión de consulta genera la secuencia personasPor Año con las personas de la lista listPersona agrupadas por años:

```
var personasPorAño =
       from p in listPersona
       group p by p.FechaNac.Year; // clave: año

foreach (var grupoPersonas in personasPorAño)
{
        Console.WriteLine(grupoPersonas.Key); // año
       foreach (var persona in grupoPersonas)
       Console.WriteLine("  {0}", persona.Nombre);
}
```

El resultado es una secuencia de elementos de tipo IGrouping<TKey, T> (hereda de IEnumerable<T>) que define una propiedad del tipo de la clave de agrupación, en nuestro caso de tipo int.

Este otro ejemplo, haciendo uso de la lista de países y de la de personas, agrupa las personas según su país de nacimiento:

```
var personasPais =
from pais in listPais
       join pers in listPersona on pais.Codigo equals pers.PaisNac
group new { Nombre = pers.Nombre } by pais.Nombre;

foreach (var grupoPerPais in personasPais)
{
Console.WriteLine(grupoPerPais.Key); // nombre pais
Foreach (var persona in grupoPerPais)
       Console.WriteLine("  {0}", persona.Nombre);
}
```

Productos cartesianos

En bases de datos, el producto cartesiano de dos tablas no es más que otra tabla resultante de combinar cada fila de la primera con cada fila de la segunda. En LINQ podemos aplicar esta definición utilizando dos clausulas from. Por ejemplo:

```
var productoCartesiano =
       from pais in listPais
       from pers in listPersona
select new { NomPais = pais.Nombre, NomPers = pers.Nombre };

foreach (var elem in productoCartesiano)
Console.WriteLine("{0}  {1}", elem.NomPers, elem.NomPais);
```

Es recomendable evitar los productos cartesianos por la explosión combinatoria que generan, o crearlos con restricciones. Por ejemplo, la siguiente expresión de consulta muestra el nombre de las personas junto con el país donde nacieron:

```
var productoCartesiano =
from pais in listPais
       from pers in listPersona
where pais.Codigo == pers.PaisNac
       select new { NomPers = pers.Nombre, NomPais = pais.Nombre };


foreach (var elem in productoCartesiano)
Console.WriteLine("{0}  {1}", elem.NomPers, elem.NomPais);
```

Clausula into

La clausula into puede utilizarse para crear un identificador temporal que almacene los resultados de una clausula group, join o select en un nuevo identificador. Esto se realiza cuando deba realizar operaciones de consulta adicionales sobre una consulta después de una operación de agrupación o selección. Por ejemplo, la siguiente expresión de consulta genera una secuencia persNacidosPorAño con los años de nacimiento de las personas de la lista listPersona agrupadas por año de nacimiento más el número de personas de cada grupo, pero solo los grupos con un número mínimo de personas:

```
var persNacidasPorAño =
from p in listPersona
group p by p.FechaNac.Year into grupoPersAño
where grupoPersAño.Count() >= 2
       select new { Año = grupoPersAño.Key, PorAño = grupoPersAño.Count()
};
```

```
foreach (var persona in persNacidasPorAño)
{
Console.WriteLine("En {0} nacieron {1} o más personas.", persona.Año,
persona.PorAño);
}
```

La siguiente expresión de consulta podemos modificarla para que los países aparezcan en orden ascendente:

```
var personasPais =
       from pais in listPais
join pers in listPersona on pais.Codigo equals pers.PaisNac
group new { Nombre = pers.Nombre } by pais.Nombre;
```

Para ello es necesario introducir ese resultado en una secuencia y ordenándola. Utilizaremos into con group. Esto es:

```
var persPais =
       from pais in listPais
join pers in listPersona on pais.Codigo equals pers.PaisNac
group new { Nombre = pers.Nombre }
by pais.Nombre into persPaisOrd orderby persPaisOrd.Key
select persPaisOrd;
```

Otro ejemplo. ¿Cómo obtenemos una lista de países ordenada ascendentemente con el número de personas por país? Combinamos cada país con las personas de ese país y las contamos. Para ello utilizaremos into con join. Esto es:

```
var PaisNumPers =
from pais in listPais
orderby pais.Nombre
join pers in listPersona on pais.Codigo equals pers.PaisNac into grupoPais
select new { NomPais = pais.Nombre, NumPers = grupoPais.Count() };

foreach (var pais in PaisNumPers)
{
Console.WriteLine("En {0} hay {1} personas.", pais.NomPais, pais.NumPers);
}
```

Clausula join

La clausula join se utiliza para realizar una operación de combinación (join funciona siempre con colecciones de objetos, en lugar de tablas de base de datos, aunque en LINQ no es necesario utilizar esta clausula tan a menudo como en SQL, porque las claves externas en LINQ se representan en el modelo de objetos como propiedades que contienen una colección de elementos) para asociar y/o combinar elementos de un origen de datos con elementos de otro origen de datos según una comparación de igualdad entre claves especificadas en cada elemento.En LINQ, las operaciones de combinación se realizan sobre secuencias de objetos cuyos elementos son de tipos diferentes.Después de haber unido dos secuencias, deberá utilizar una instrucción select o group para especificar qué elemento va a almacenar en la secuencia de salida.

Con join se trata de limitar las combinaciones que produciría un producto cartesiano, manteniendo únicamente los elementos de las secuencias que coinciden con el criterio establecido. Por ejemplo:

```
var combinacion =
from pais in listPais
join pers in listPersona on pais.Codigo equals pers.PaisNac
select new { NomPers = pers.Nombre, NomPais = pais.Nombre };

foreach (var elem in combinacion)
Console.WriteLine("{0}  {1}", elem.NomPers, elem.NomPais);
```

Clausula let

La clausula let se utiliza para almacenar el resultado de una subexpresión con el fin de utilizarlo en clausulas posteriores. Por ejemplo, la siguiente expresión de consulta, utilizando la clausual let, genera una secuencia persPorAñoMes con las personas que han nacido en un mes y un año determinados:

```
int unMes = 12, unAño = 1987;

var persPaisAñoMes =
from pais in listPais
orderby pais.Nombre
join pers in listPersona on pais.Codigo equals pers.PaisNac
let mesNac = pers.FechaNac.Month
let añoNac = pers.FechaNac.Year
where añoNac == unAño && mesNac == unMes
group new { Nombre = pers.Nombre } by pais.Nombre into persPais
select persPais;

Console.WriteLine("Nacidos en el mes {0} del año {1}:", unMes, unAño);
foreach (var grupoPerPais in persPaisAñoMes)
{
Console.WriteLine(grupoPerPais.Key);
foreach (var persona in grupoPerPais)
Console.WriteLine(" {0}", persona.Nombre);
}
```

PROVEEDORES DE LINQ

En LINQ distinguimos proveedores locales y proveedores remotos. Los proveedores locales son aquellos que operan sobre objetos en memoria. Estos objetos tienen que implementar la interfaz IEnumerable<T>. A esta categoría pertenecen LINQ to Objects, LINQ to DataSet y LINQ to XML. Y los proveedores remotos son aquellos que operan sobre bases de datos relacionales. En este caso, el mecanismo basado en el recorrido de colecciones de objetos en memoria (más bien secuencias) que tan bien nos han funcionado en los proveedores locales, no resulta adecuado en los proveedores remotos, simplemente porque cualquier implementación de un operador de consulta basada en la navegación de cursores (un cursor puede verse como un iterador sobre la colección de filas de un conjunto de datos obtenido después de una consulta SQL) sería un fracaso desde el punto de vista de rendimiento. Debido a esto, se definió la interfaz IQueryable<T>, que hereda de IEnumerable<T>, que es la que implementan los proveedores LINQ to SQL y LINQ to Entities. Quiere esto decir que los objetos que implementen la interfaz IQueryable<T> pueden ser también orígenes de consultas integradas, pero lo que verdaderamente aporta esta interfaz es un conjunto de métodos extensores con una implementación más adecuada para interaccionar con bases de datos relaciones que la proporcionada por los métodos de LINQ to Objects.

Los proveedores LINQ to Objects y LINQ to SQL, ambos proporcionan prácticamente los mismos operadores de consulta. La diferencia está en que los métodos extensores de LINQ to SQL en lugar de recibir delegados como parámetros, reciben arboles de expresiones. No obstante, esto no es un problema ya que una expresión lambda puede transformarse en un delegado y también en un árbol de expresión.

Todos los métodos incorporan una referencia a un árbol de expresión que define el algoritmo de obtención de la secuencia. Como ejemplo, observe la firma del método Where aportado por la interfaz IQueryable<T>:

> Public static IQueryable<TOrigen> Where <TOrigen> (this I Queryable<TOrigen> origen, Expression<Func<TOrigen, bool>> predicado)

Observemos que este método tiene al menos un parámetro de tipo Expression<TDelegado> cuyo argumento de tipo es un delegado Func. Por lo tanto, es posible pasar una expresión lambda y compilarla en un objeto Expression<TDelegado>. Finalmente, la consulta que se obtiene como resultado de ejecutar un árbol de expresión que representa al método Where que realiza la llamada devolverá los elementos de *origen* que satisfagan la condición especifica en predicado.

LINQ to SQL

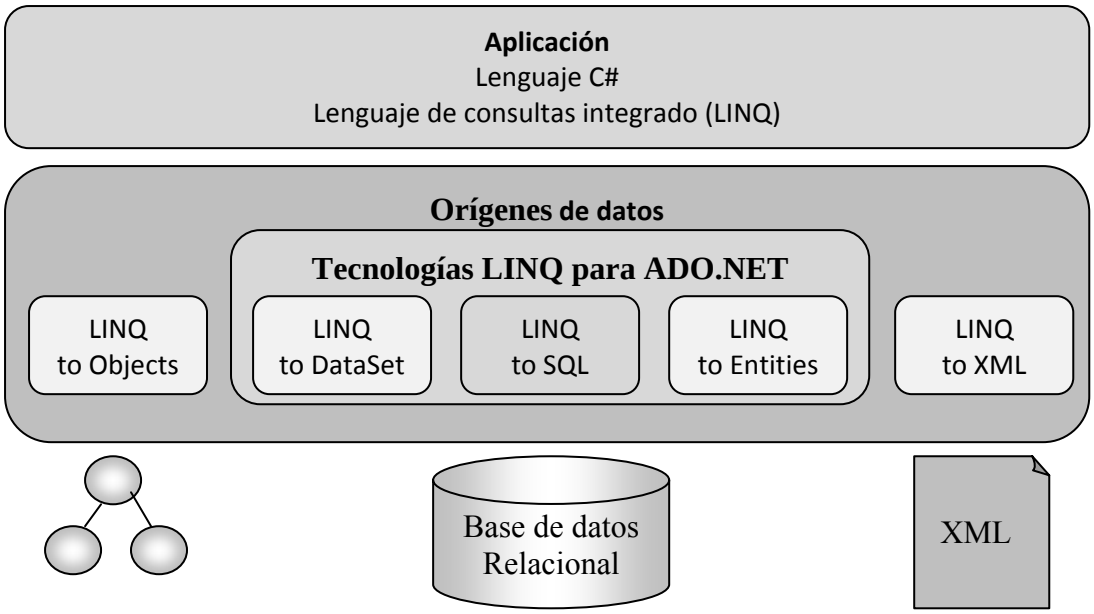
LINQ nos ofrece la posibilidad de expresar las operaciones de consulta en el propio lenguaje y no como literales de cadena pertenecientes a otro lenguaje incrustados en el código de aplicación (por ejemplo, sentencias SQL). Y, ¿cómo se puede con LINQ realizar consultas a una base de datos? Pues utilizando el proveedor LINQ to SQL para asignar, durante el diseño, el modelo de datos relacional de una base de datos a un modelo de objetos expresado en el lenguaje de programación del programador, bien manualmente o mediante el diseñador relacional de objetos, y realizar después las consultas sobre el modelo de objetos. LINQ to SQL convierte a SQL estas consultas integradas en el lenguaje y las envía a la base de datos para su ejecución. Cuando la base de datos devuelve los resultados, LINQ to SQL los vuelve a convertir en objetos expresados en el propio lenguaje de programación utilizado.

Hay una gran diferencia entre LINQ to SQL y LINQ. LINQ Es un lenguaje de consulta integrado que se puede ejecutar sobre varias fuentes por medio de los distintos proveedores de LINQ hasta la fecha desarrollados. Estas fuentes y los proveedores correspondientes son:

- DataSet: LINQ to DataSet.
- XML: LINQ to XML.
- Objetos de memoria: LINQ to Objects.
- Bases de datos relacionales: LINQ to SQL y Entity Framework

Todos estos proveedores, excepto Entity Framework se lanzaron con .NET Framework 3.5.

LINQ to SQL es el proveedor de datos que LINQ ofrece para MS SQL Server. La figura siguiente nos da una idea de donde actúa este proveedor.

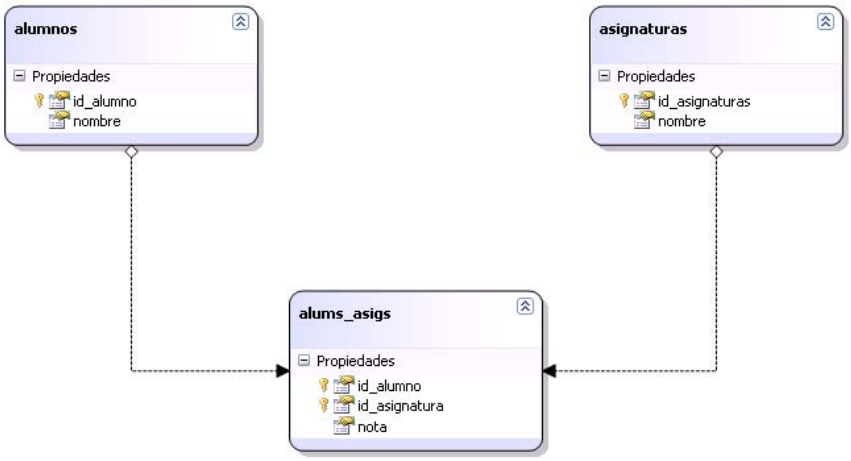


Cuando utilizamos el proveedor LINQ to SQL para realizar consultas a una base de datos relacional el modelo de datos de esa base de datos se asigna, durante el diseño, a un modelo de objetos expresado en el lenguaje de programación del programador, ya sea manualmente o mediante el *diseñador relacional de objetos* (diseñador R/O). En la práctica esto crea una base datos orienta a objetos virtual sobre la base de datos

relacional. Esto posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo). Después se escriben las consultas en el modelo de objetos y cuando la aplicación se ejecuta, LINQ to SQL convierte a SQL la consulta integrada en el lenguaje y las envía a la base de datos para su ejecución. Cuando la base de datos devuelve los resultados, LINQ to SQL los vuelve a convertir en objetos con los que pueda trabajar en su propio lenguaje de programación.

ACCESO A UNA BASE DE DATOS

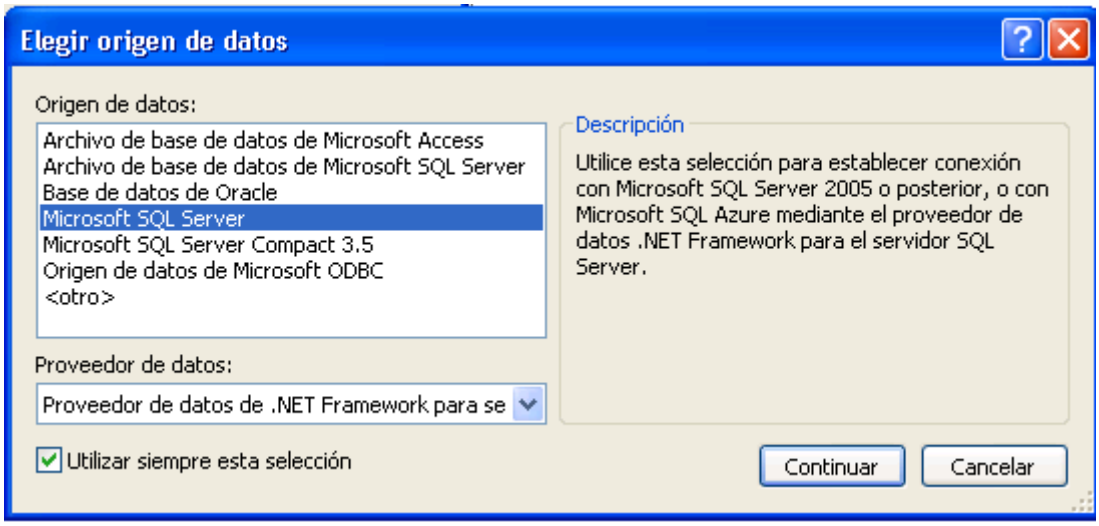
Se realizará una aplicación que trabaje con los datos de una base de datos SQL Server para mostrar el acceso a una base de datos. Para ello se crearála base de datos *bd_notasAlumnos*, la cual almacena las notas de los alumnos matriculados.



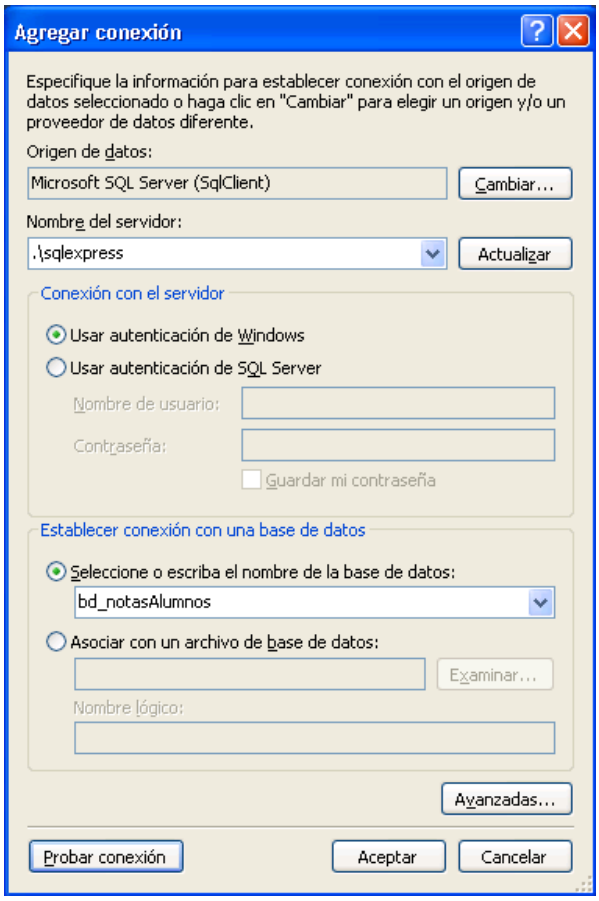
Primeramente se crea una aplicación con Visual Studio denominada *ApLinq*. Para ello se crea un nuevo proyecto de tipo *Visual C# > Windows> Aplicación de Windos Forms*.

Añadir una conexión con la base de datos

Para añadir la conexión con la base de datos ver > *explorador de servidores>conectar con base de datos*. Luego elegimos el origen de datos.



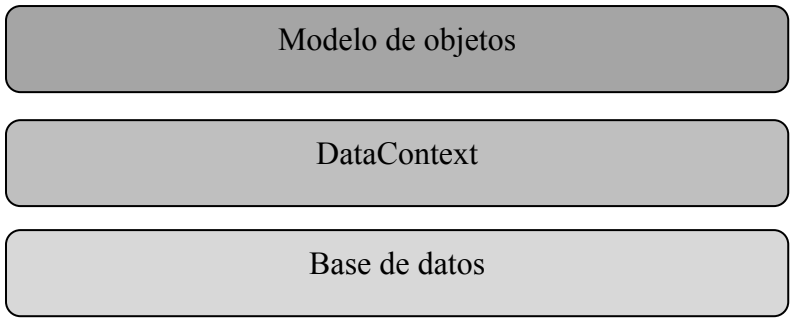
Añadimos la conexión a la base de datos *bd_notasAlumnos* de la siguiente manera:



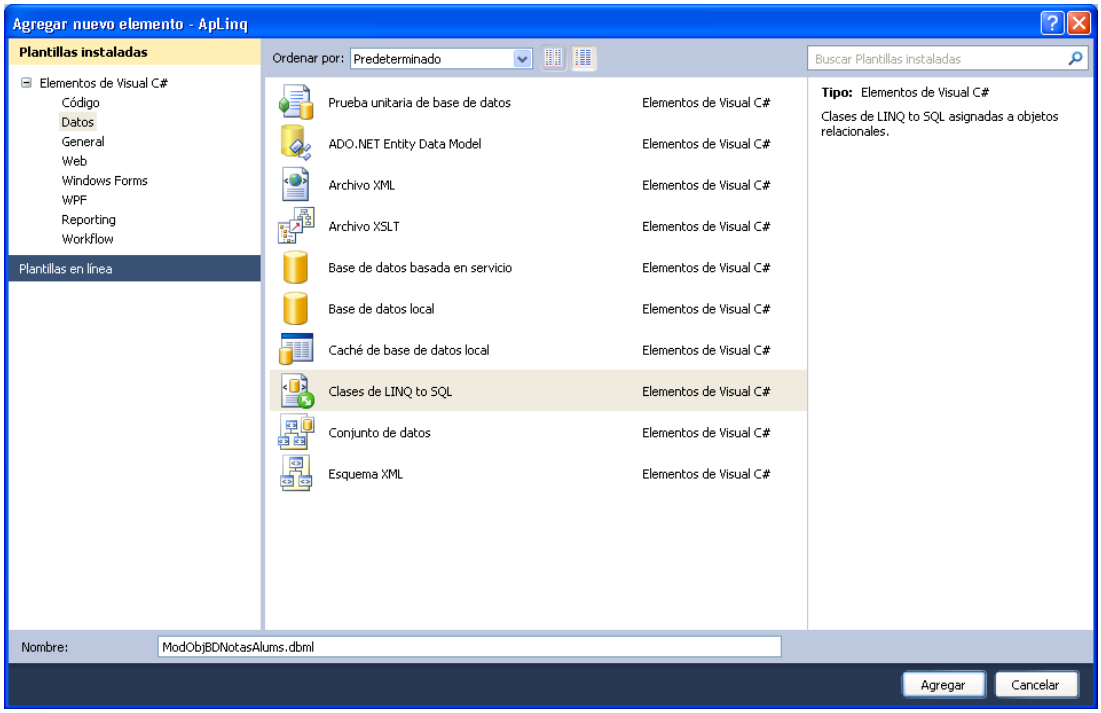
Generar el modelo de objetos

Un modelo de objetos expresado en el lenguaje de programación con el que se esté escribiendo la aplicación representa una base de datos orientada a objetos virtual imagen del modelo de datos de una base de datos relacional. Para generar automáticamente este modelo a partir de una base de datos existente utilizaremos el *diseñador relacional de objetos* de visual estudio.

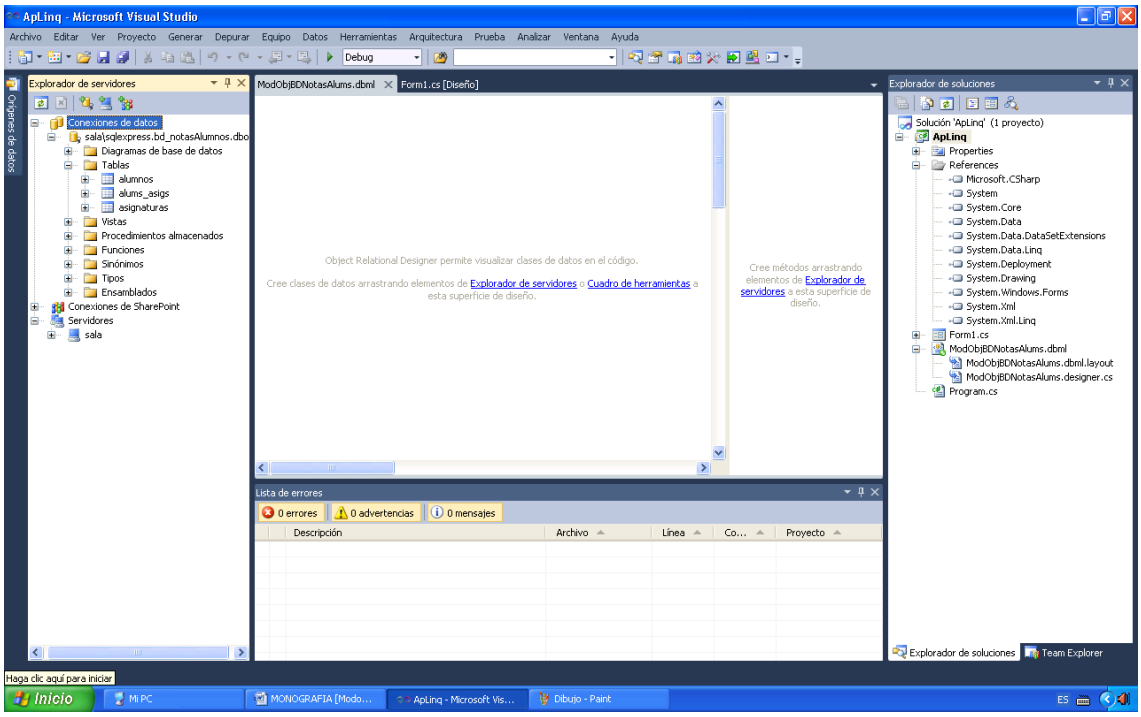
El diseñador relacional de objetos proporciona una superficie de diseño visual para crear clases de entidad y asociaciones (relacionales) de LINQ to SQL basada en los objetos de una base de datos. El resultado es un modelo de objetos representativo de los objetos de una base de datos. También, el diseñador genera una clase derivada de DataContext que facilita las operaciones de enviar y recibir datos entre las clases de entidad y la base de datos, proporciona la funcionalidad necesaria para asignar los procedimientos almacenados y funciones a los métodos de DataContext con el fin de devolver datos y rellenar las clases de entidad,también permite diseñar relaciones de herencia entre las clases de entidad.



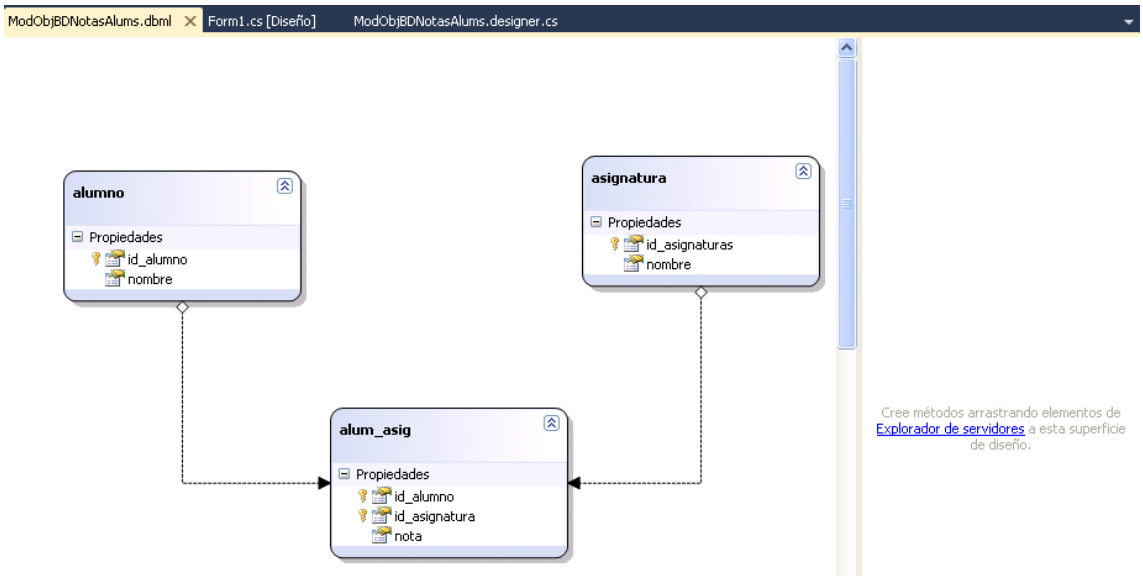
Para abrir el diseñador relacional de objetos agregamos un nuevo elemento de tipo Clases de LINQ to SQL al proyecto. Lo denominaremos *ModObjBDNotasAlums.dbml*.



Después de hacer clic en el botón *Agregar*, la ventana de diseño de Visual Studio mostrara el diseñador relacional de objetos que muestra la figura siguiente:



En los paneles del diseñador observamos que podemos crear clases de datos y métodos arrastrando elementos desde la conexión de la base de datos que tenemos abierta en el explorador de bases de datos. Seleccionamos en el explorador de bases de datos todas las tablas con las que vaya a trabajar (*alumnos, asignaturas y alums_asigs*) y las arrastramos sobre la superficie de diseño. Observamos que se muestran las clases de entidad que están asignadas a las tablas de la base de datos y sus relaciones:



Para facilidad en la comprensión del desarrollo se sugiere que el nombre de las clases de entidad este en singular, por ejemplo *alumno*, ya que representa una fila de la tabla de la base de datos, en este caso de la tabla *alumnos*.

Al arrastrar las tablas se crearon automáticamente las asociaciones entre ellas basándose en las relaciones de clave externa existentes en la base de datos. No obstante, también

sepuede realizar esta asociación haciendo clic sobre la clase de entidad y seleccionando del menú contextual la opción *Agregar Asociación*.

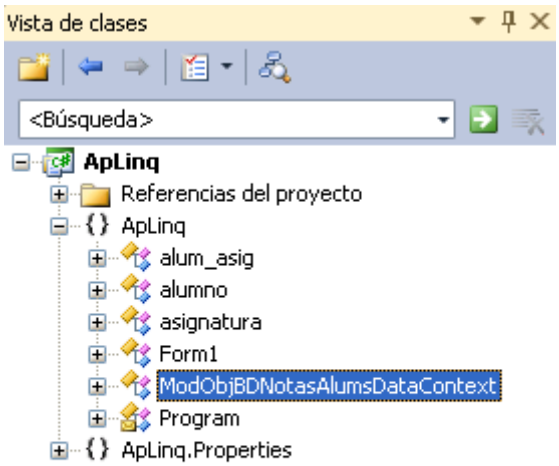
De forma predeterminada, LINQ to SQL crea la lógica para actualizar una base de datos (inserciones, actualizaciones y eliminaciones) con los cambios realizados en los datos de las clases de entidad, basándose en el esquema de la tabla (información de columna y de clave principal). Cuando no se desee usar el comportamiento predeterminado, se puede configurar el comportamiento asignando procedimientos almacenados concretos a través del diseñador.

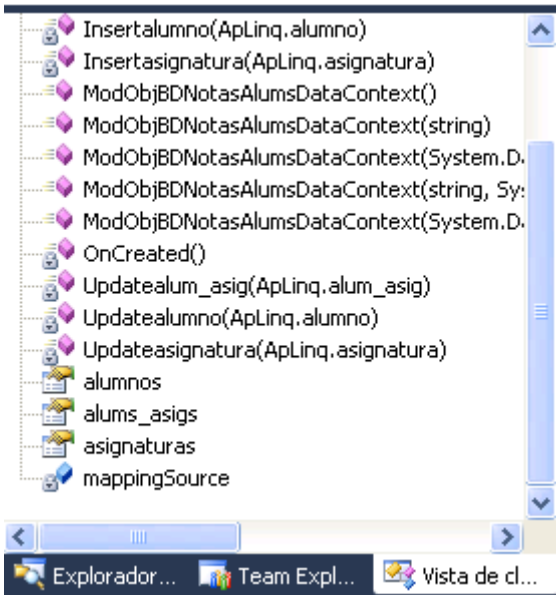
El panel de métodos, situado a la derecha y que de forma predeterminad esta vacio, mostrara aquellos métodos de DataContext que creemos arrastrando desde el explorador de servidores/explorador de bases de datos hasta el diseñador relacional de objetos los procedimientos almacenados o funciones que consideremos necesarios. El tipo del valor devuelto de cada método generado difiere según se coloque el elemento en una clase de entidad existente, en este caso el valor devuelto coincide con un objeto de la clase de entidad, o en un áreavacía, en cuyo caso se crea un método que devuelve un objeto de un tipo que se genera automáticamente, de nombre el del procedimiento almacenado o de la función, y cuyas propiedades se asignan a los campos devueltos por el procedimiento almacenado o la función. No obstante, una vez agregado un método, podemos cambiar el tipo del valor devuelto a través de la ventana de propiedades. Estos métodos a veces se añaden con el fin de invalidar el comportamiento predeterminado de LINQ to SQL para realizar inserciones, actualizaciones y eliminaciones, como consecuencia de invocar al método SubmitChanges para guardar los cambios de las clases de entidad en una base de datos.

Hasta el momento hemos generado el modelo de objetos y el contexto de datos (DataContext) necesario para enviar y recibir datos entre las clases de entidad y la base de datos.

Las clases de entidad y el contexto de datos

En la figura siguiente observamos que se ha generado una clase de entidad por cada tabla que hemos arrastrado de la base de datos y otra clase *ModObjBDNotasAlumnsDataContext* derivada de DataContext. Las clases de entidad constituyen el modelo relacional de objetos y la otra, el contexto de datos que actúa como intermediario entre las clases de entidad y la base de datos SQL Server. A su vez, cada clase de entidad define una propiedad por cada columna de la tabla asociada, con el mismo nombre, así como otras propiedades para definir las relaciones que existen entre las tablas (estas devuelven un EntitySet<*clase_entidad*> o un EntityRef<*clase_entidad*>). Todas las clases han sido definidas en el fichero Mod-*ObjBDNotasAlums.designer.cs*;





El código de las clases de entidad puede ser extendido, por ejemplo, si deseamos validar los datos antes de enviar las actualizaciones a la base de datos subyacente. Para ello, hacemos clic con el botón secundario del ratón en la clase sobre la que se desee agregar la validación y, luego, hacemos clic en ver código.

La clase DataContext contiene la información de la cadena de *conexión* la cual es proporcionada a través de su constructor.También contiene varias propiedades, por ejemplo, *alumnos*, y varios *métodos*, por ejemplo, el método SubmitChanges que envía los datos actualizados de las clases de LINQ to SQL a la base de datos, a los que se puede llamar para realizar las operaciones deseadas sobre la base de datos. También, como ya dijimos anteriormente, se pueden crear nuevos métodos asociados con los procedimientos almacenados y funciones. Fíjese también, en la figura anterior, en las definiciones parciales de los métodos, por ejemplo, *Insertasignatura, Updateasignatura o Deleteasignatura* (si implementa estos métodos en su clase parcial, el motor de ejecución de LINQ to SQL los llamará al llamar a SubmitChanges en lugar de llamar a los ya implementados por el sistema; esta es la alterativa que tenemos para invalidar, cuando sea necesario, el comportamiento predeterminado), o en las

propiedades*alumnos, asignaturas o alums_asigs* que retornan la tabla correspondiente, dicho de otra forma, retornan una secuencia de objetos "filas de la tabla" encapsulados en un objeto de tipo Table<*clase_entidad*> que implementa la interfaz IQueryable que hereda de IEnumerable por lo que podremos aplicar LINQ sin problemas. Por ejemplo, la propiedad *alumnos* devuelve un objeto Table<*alumno*> que se corresponde con una secuencia de objetos de la clase de entidad *alumno*, cada uno de los cuales se corresponde con una fila de la tabla *alumnos*.

Según lo expuesto, para que una aplicación dotada de un contexto de datos, como lo es *ModObjBDNotasAlumsDataContext*, pueda acceder a la base de datos correspondiente, deberá crear un objeto DataContext, como se muestra a continuación:

```
       ModObjBDNotasAlumsDataContext contextoDeDatos =
            new ModObjBDNotasAlumsDataContext();
            // Obtener la tabla alumnos
            Table<alumnos> TablaAlumnos = contextoDeDatos.GetTable<alumnos>();
            // Realizar una consulta
            var alums = from alum in TablaAlumnos
                        where alum.id_alumno == 1234567
                        select new { Nombre = alum.nombre };
if (alums.Count() > 0)
               MessageBox.Show(alums.First().Nombre);
```

¿Cuánto tiempo permanece abierta la conexión con la base de datos? Normalmente, una conexión permanece abierta mientras se utilizan los resultados de la consulta. Por ejemplo:

```
       var alums = from alum in contextoDeDatos.GetTable<alumno>()
       select alum
       Console.WriteLine(contextoDeDatos.Connection.State);//escribe: Closed
       foreach (var alum in alums)
       {
              // ...
              Console.WriteLine(contextoDeDatos.Connection.State);//escribe: Open
       }
       Console.WriteLine(contextoDeDatos.Connection.State); // escribe: Closed
```

Ahora bien, si espera que los resultados van a tardar bastante tiempo en procesarse, y no le causa ningún problema que se almacenen en memoria cache, aplique el método ToList a la consulta:

```
       foreach (var alum in alums.ToList())
       {
              // ...
              Console.WriteLine(contextoDeDatos.Connection.State);//escribe:
Closed
       }
```

Analicemos ahora una clase de entidad; por ejemplo, la clase de entidad *alumno* que está asociada a la tabla *alumnos* de la base de datos:

```
       [Table(Name="dbo.alumnos")]
       public patrial class alumnos:     INotifyPropertyChanging,
                           INotifyPropertyChanged
       {
              private int _id_alumno;
```

```
            private string _nombre;
            private EntitySet <alum_asig> _alums_asigs;
            // ...

            [Column(Storage="_id_alumno",      DbType="Int      NOT      NULL",
            IsPrimaryKey=true)]

            public int id_alumno
            {
                  get
                  {
                        return this._id_alumno;
                  }
                  set
                  {
                        // ...
                  }
            }
            [Column(Storage="_nombre",     DbType="VarChar(50)     NOT     NULL",
            CanBeNull=true)]
            public string nombre
            {
                  // ...
            }

            [Association(Name="alumnos_alums_asigs",     Storage="_alums_asigs",
            ThisKey="id_alumno", OtherKey="id_alumno")]
            public EntitySet <alum_asig> alums_asigs
            {
                  get
                  {
                        return this._alums_asigs;
                  }
                  set
                  {
                        this._alums_asigs.Assign(value);
                  }
            }
            // ...
      }
```

En el código anterior observamos la clase de entidad *alumno* que está asociada a la tabla alumnos de la base de datos. Esta clase, que representa una fila de la tabla, define por cada campo de la tabla *alumnos* (clausula Association) una propiedad del mismo nombre, *id_alumno y nombre*, que devuelve el valor correspondiente al atributo vinculado con la misma y una propiedad más, *alumns_asigs* que define la relación entre las tablas, en este caso, entre *alumnos* y *alums_asigs* (en la primera *id_alumno* es la clave primaria, que, a su vez, es la clave externa en la segunda), que devuelve un objeto EntitySet<*alum_asig*> que se corresponde con una secuencia de objetos *alum_asig*. El ejemplo siguiente muestra como gracias a esta última propiedad es posible acceder a las notas de las asignaturas de un alumno de una forma muy simple:

```
      ModObjBDNotasAlumsDataContext contextoDeDatos =
            new ModObjBDNotasAlumsDataContext();

            var actas =
from alum in contextoDeDatos.alumnos
from al_as in alum.alum_asig
               group new { alum.nombre, al_as.nota } by al_as.id_asignatura;
```

```
            foreach (var grupo in actas)
            {
                MessageBox.Show("\nAsignatura: " + grupo.Key);
foreach (var al in grupo)
                    MessageBox.Show(al.nombre + " " + al.nota);
            }
```

La expresión *alum.alums_asigs* devuelve un objeto Entity<*alum_asig*> con las asignaturas y notas del alumno *alum*.

La propiedad *alums_asigs* que define la relación entre las tablas *alumnos y alums_asigs* permite para un objeto *alumno* acceder a los datos que figuran de este objeto en la tabla relacionada *alums_asigs* (partimos desde una fila de la tabla con clave primaria y finalizamos en la tabla relacionada con la misma clave pero externa).

Observamos la simplificación de la expresión de consulta *from al_as in alum.alums_asigs* respecto a esta otra, que da el mismo resultado:

```
join al_as in contextoDeDatos.alums.asigs
       on alum.id_alumno equals al_as.id_alumno
```

La clase de entidad asignatura es similar a la clase de entidad alumno. A continuación se muestra la clase de entidad*alum_asig*:

```
[Table(Name="dbo.alums_asigs")]
public patrial class alum_asig:   INotifyPropertyChanging,
                            INotifyPropertyChanged
{
       private int _id_alumno;
       private int _id_asignatura;
       private float _nota;

       private EntityRef <alumno> _alumno;
       private EntityRef <asignatura> _asignatura;
       // ...

       [Column(Storage="_id_alumno",      DbType="Int      NOT      NULL",
       IsPrimaryKey=true)]

       public int id_alumno
       {
              // ...
       }
       [Column(Storage="_id_asignatura",   DbType="Int   NOT   NULL",   I
       sPrimaryKey=true)]
       public int id_asignatura
       {
              // ...
       }
       [Column(Storage="_nota", DbType="Real NOT NULL")]
       public float nota
       {
              // ...
       }

       [Association(Name="alumnos_alums_asigs",          Storage="alumno",
       ThisKey="id_alumno", OtherKey="id_alumno", IsForeignKey=true)]
       public alumno alumno
       {
```

```
                get
                {
                        return this.alumno.Entity;
                }
                set
                {
                        // ...
                }

        }

        [Association(Name="asignaturas_alums_asigs",   Storage="asignatura",
        ThisKey="id_asignatura",                  OtherKey="id_asignatura",
        IsForeignKey=true)]
        public asignatura asignatura
        {
                get
                {
                        return this._asignatura.Entity;
                }
                set
                {
                        // ...
                }
        }
        // ...
}
```

Observamos que la clase de entidad *alum_asig*, que está asociada a la tabla *alums_asigs* de la base de datos, también define una propiedad por cada campo de la tabla y, además, dos propiedades más: *alumno y asignatura*, que definen las relaciones de la tabla *alums_asigs* con la tabla *alumnos y asignaturas* (en la primera *id_alumno e id_asignatura* son claves externas y, respectivamente, en la segunda son claves primarias); estas dos propiedades devuelven un objeto *clase_entidad (alumno o asignatura*) a través de un atributo de tipo EntityRef<*clase_entidad*> (entidad en el extremo de la relación de multiplicidad uno). El ejemplo siguiente muestra como gracias a estas dos propiedades es posible acceder a las asignaturas de las que se ha matriculado un alumno de una forma muy simple:

```
ModObjBDNotasAlumsDataContext contextoDeDatos =
        new ModObjBDNotasAlumsDataContext();

int idAlumno = 1234567;

var alumAsigs =
        from al_as in contextoDeDatos.alums_asigs
where al_as.id_alumno == idAlumno
                group new { NombAsig = al_as.asignatura.nombre } by
        al_as.alumno.nombre;

foreach (var grupo in alumAsigs)
{
MessageBox.Show("\nAlumno: " + grupo.Key);
foreach (var x in grupo)
        MessageBox.Show(x.NombAsig);
}
```

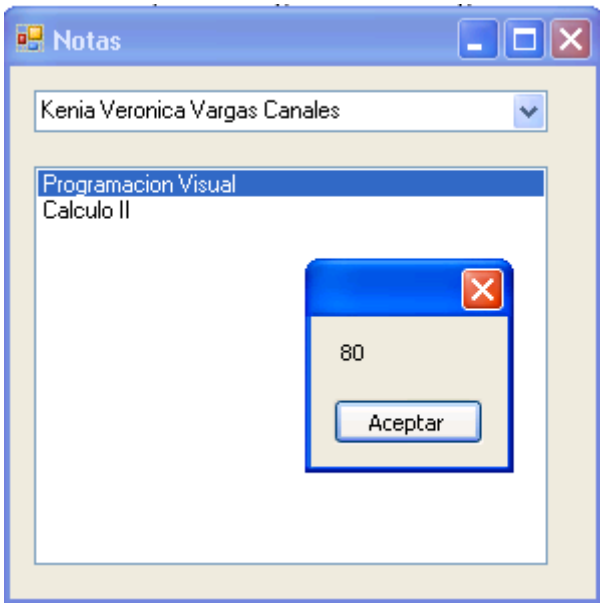
Las expresiones *al_as.asignatura y al_as.alumno* haciendo uso de EntityRef devuelven, respectivamente, un objeto asignatura y otro alumno.

Las propiedades *alumno y asignatura* que definen las relaciones entre la tabla *alums_asigs* y las tablas *alumnos y asignaturas* permiten para un objeto *alum_asig* acceder a los datos que figuran de este objeto en la tabla relacionada *alumnos o asignaturas* (partimos desde una fila de la tabla con clave externa y finalizamos en la tabla relacionada con la misma clave pero primaria).

Crear un origen de datos

Una vez generado el modelo de objetos, las clases de entidad y el contexto de datos veremos cómo trabaja la aplicación contra la base de datos utilizando este modelo de objetos. Para ello generamos un formulario llamado "Notas". El cual, inicialmente, mostrara una interfaz que permita a un alumno acceder a la base de datos para conocer la nota obtenida en cualquiera de las asignaturas en las que está matriculado.

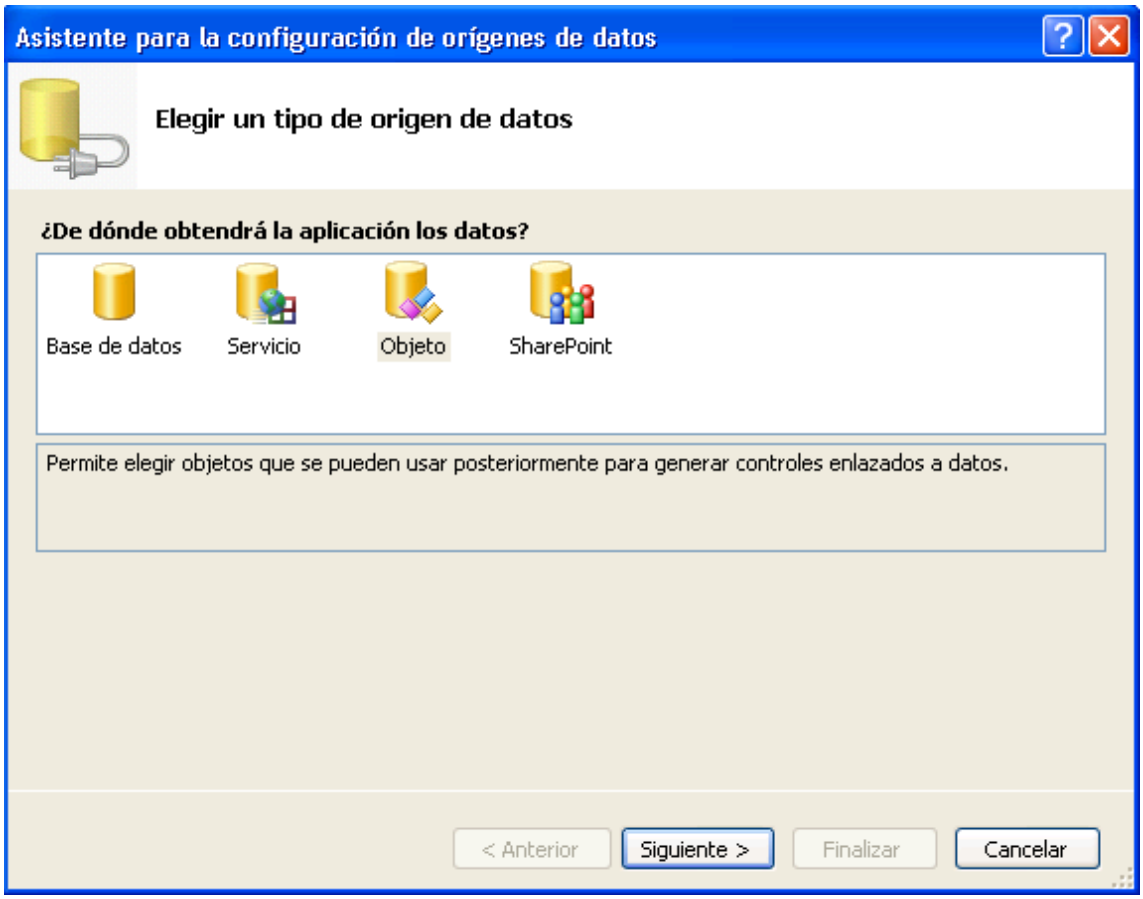
El alumno seleccionará su nombre en la lista desplegable (este control muestra los nombres de la tabla *alumnos*) lo que hará que se muestren en la lista fija las asignaturas en las que está matriculado. Después, seleccionará la asignatura de la cual quiere conocer la nota y esta le será mostrada en una caja de diálogo.



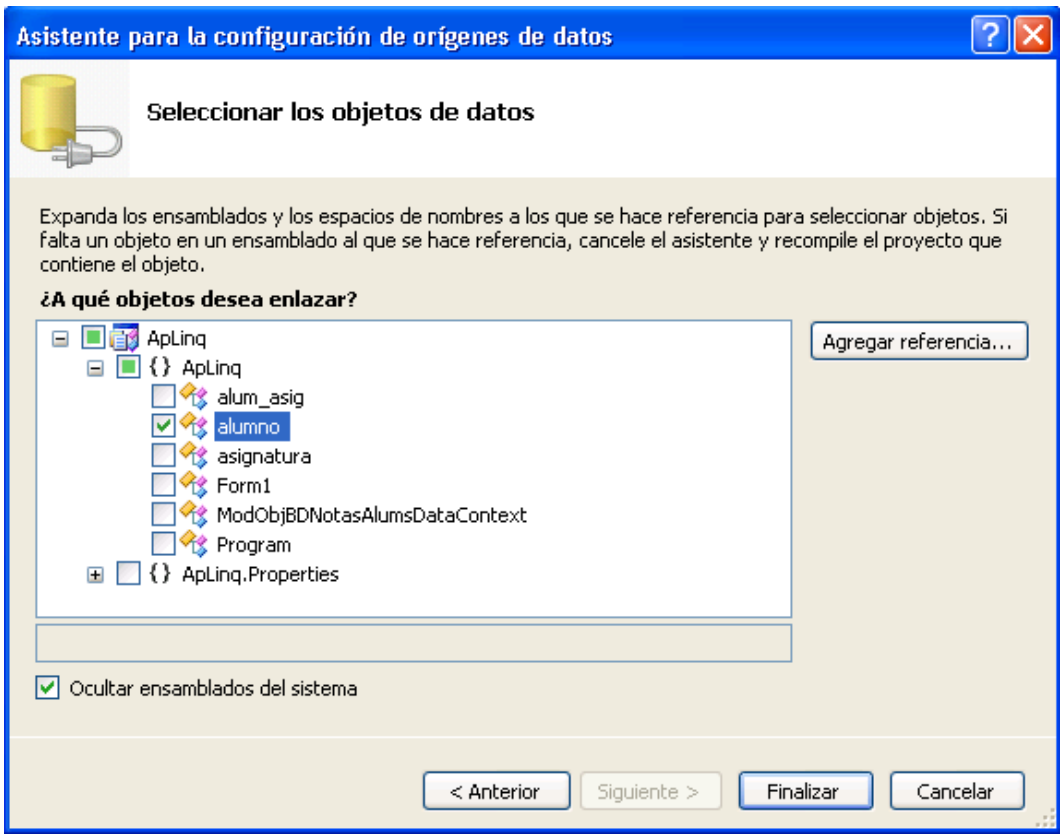
Ya tenemos generado el proyecto y el modelo de objetos (representación de la base de datos). Partiendo de este modelo, creamos un origen de datos que haga referencia a la clase de entidad *alumno*.

Una clase de entidad, al igual que cualquier otra clase con propiedades públicas, se puede usar como un origen de datos de tipo *Objeto*. Se puede agregar a la aplicación un origen de datos y arrastrarlo hasta el formulario para crear controles enlazados a datos; los datos serán los valores que almacenan las propiedades públicas del objeto.

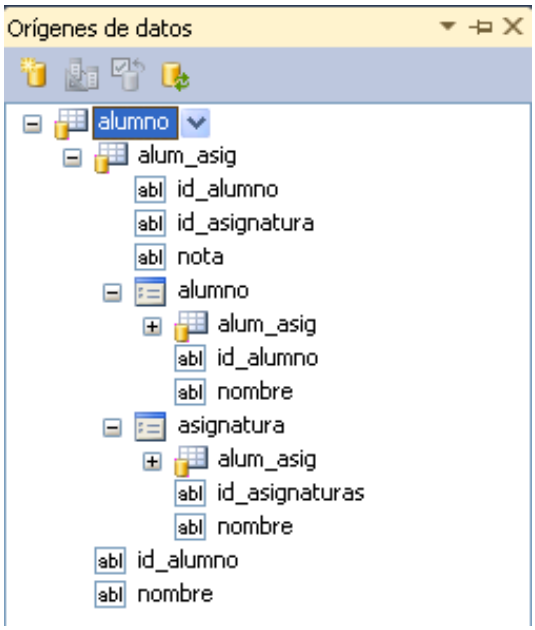
Para agregar la clase de entidad *alumno* como origen de datos de tipo *Objeto* en la ventana *Orígenes de datos*, ejecutamos la orden *Agregar nuevo origen de datos* del menú *Datos*:



Después seleccionamos*Objeto* como tipo de origen de datos y hacemos clic en *Siguiente* para poder seleccionar el objeto que va a ser origen de los datos.



Expandimos el nodo ApLinq, luego, seleccionamos la clase *alumno*. Si la clase *alumno* no está disponible, cerramos el asistente, generamos el proyecto y volvemos a ejecutar el asistente. Después, hacemos clic en *Siguiente* y en *Finalizar*, para crear el origen de datos desde la clase de entidad *alumno* y agregarlo a la ventana *Orígenes de datos*. El resultado se observa en la figura siguiente:



No solo se muestra la clase *alumno*, sino también su relación con las otras clases a través de su propiedad *alums_asigs* que define la relación *alumnos_alums_asigs*.

Vincular controles con el origen de datos

A continuación, añadimos al formulario un ComboBox, *listaDesAlumnos*, y lo vinculamos con el origen de datos *alumno*. Este control tiene que mostrar el *nombre* del alumno y guardar como valor el *id_alumno*.

Configuramos este control para vincularlo con el origen de datos que hemos creado. Para ello, desde la ventana de propiedades, asignamos a su propiedad DataSource el origen de datos *alumno* (esta acción genera el objeto alumnoBindingSource de tipo BindingSource que hace de intermediario entre el control y el origen de datos y lo asigna como origen de datos definitivo a DataSource) y después asignamos a su propiedad DisplayMember al campo nombre y a ValueMember el campo id_alumno.

Si ejecutamos la aplicación notaremos que la lista aun no muestra nada.Porque el origen de datos de la lista es ahora *alumnoBindingSource* y la propiedad *DataSource* de este no tiene, a su vez, asignado el origen de datos *alumno*, sino que actualmente tiene este valor:

```
listaDesAlumnos.DataSource = alumnoBindingSource;
// ...
alumnoBindingSource.DataSource = typeof(ApLinq.alumno);
```

El operador typeof de C# devuelve un objeto Type que encapsula la información del tipo especificado; esto nos permite utilizar los miembros de Type para obtener esa

información: constructores, métodos, campos, propiedades y eventos. La información de *alumno* permite durante el diseño a las propiedades de *listaDesAlumnos*, como *DisplayMember* y *ValueMember*, conocer de que propiedades de *alumno* se pueden proveer. Durante la ejecución, el valor que en realidad hay que asignar es un objeto que represente a la tabla *alumnos* de la base de datos yese objeto es devuelto por la propiedad *alumnos* del contexto de datos.Por lo tanto, tenemos que crear el contexto de datos. Para ello añadimos la siguiente línea como miembro privado de *Form1*:

```
private ModObjBDNotasAlumsDataContext contextoDeDatos = new
ModObjBDNotasAlumsDataContext();
```

Y a continuación, asignamos a la propiedad *alumnoBindingSource.DataSource* la tabla alumnos. Esto lo podemos hacer en el constructor de *Form1* o en el controlador del evento Load.

```
public Form1()
{
InitializeComponent();
alumnoBindingSource.DataSource = contextoDeDatos.alumnos;
}
```

Ahora cuando se ejecute la aplicación, la lista mostrará los nombres de los alumnos de la tabla *alumnos*. De forma resumida, ¿Qué es lo que ha sucedido? Pues algo similar a lo que especifica el código siguiente:

```
// Crear la conexión para acceder a la base de datos
string sconexion = "Data Source=.\\sqlexpress;Initial " +
       "Catalog=bd_notasAlumnos;Integrated Security=true";
SqlConnection conexion = new SqlConnection (sconexion);

// Contexto de datos configurado con la conexión
DataContext dc = new DataContext(conexion);
//Obtener la tabla alumnos
Table<alumno> TablaAlumnos = dc.GetTable<alumno> ();
// Consulta
var alums = from alum in TablaAlumnos
select alum;
// Configurar el control ComboBox
listaDesAlumnos.DisplayMember = "nombre";
listaDesAlumnos.valueMember = "id_alumno";
listaDesAlumnos.DataSource = alums.ToList();
```

Este código, utilizando el contexto de datos que definimos anteriormente como atributo privado de *Form1*, puede reemplazarse por este otro que se indica a continuación, lo que muestra que la conexión con la base de datos se realiza a través de este contexto, ya que fue configurado automáticamente para ello con la información de conexión que le proporcionó el primer elemento que se arrastró sobre el *diseñador relacional de objetos*.

```
var alums = from alum in contextoDeDatos.GetTable<alumno>()
select alum;

// Configurar el control ComboBox
listaDesAlumnos.DisplayMember = "nombre";
listaDesAlumnos.ValueMember = "id_alumno";
listaDesAlumnos.DataSource = alums.ToList()
```

Siguiendo con la aplicación, cuando el usuario seleccione un nombre de la lista desplegable, en la lista fija deben mostrarse las asignaturas de la tabla *asignaturas* que ese alumno ha matriculado. A través de la siguiente expresión de consulta obtenemos las asignaturas y notas correspondientes. En la consulta hay un parámetro variable que se corresponde con el identificador del alumno seleccionado y que viene dado por el valor de la propiedad SelectValue de la lista desplegable *listaDesAlumnos*.

```
var asigsNotas =
     from al_as in contextoDeDatos.alums_asigs
     where al_as.id_alumno == (int)listaDesAlumnos.SelectedValue &&
         al_as.id_asignatura == al_as.asignatura.id_asignaturas
     select new { NombAsig = al_as.asignatura.nombre, Nota = al_as.nota };
```

Añadimos al formulario un ListBox, *listaAsignatuasNotas*, y lo vinculamos con el origen de datos *asigsNotas*. Este control tiene que mostrar el*NombAsig* de la asignatura y guardar como valor la *Nota*. Para ello, tendremos que establecer las propiedades de la lista especificadas a continuación con los valores indicados:

```
listaAsignaturasNotas.DisplayMember = "NombAsig";
listaAsignaturasNotas.ValueMember = "Nota";
listaAsignaturasNotas.DataSource = asigsNotas;
```

Cada vez que el usuario seleccione un nombre de la lista desplegable *listaDesAlumnos*, generará el evento SelectedIndexChanged de la lista. Entonces es necesario escribir el controlador para este evento de la siguiente manera:

```
private void listaDesAlumnos_SelectedIndexChanged(object sender, EventArgs
e)
{
// Si no hay un elemento seleccionado, salir
       if (listaDesAlumnos.SelectedIndex < 0) return;

       // Consulta: asignaturas y notas del alumno seleccionado
var asigsNotas =
from al_as in contextoDeDatos.alums_asigs
where al_as.id_alumno == (int)listaDesAlumnos.SelectedValue &&
         al_as.id_asignatura == al_as.asignatura.id_asignaturas
       select new { NombAsig = al_as.asignatura.nombre, Nota = al_as.nota
       };

       // Configurar la lista fija
       listaAsignaturasNotas.SelectionMode = SelectionMode.None;
listaAsignaturasNotas.DisplayMember = "NombAsig";
       listaAsignaturasNotas.ValueMember = "Nota";
       listaAsignaturasNotas.DataSource = asigsNotas;
       listaAsignaturasNotas.SelectionMode = SelectionMode.One;

 }
```

En la configuración de la lista se ha utilizado también la propiedad SelectionMode para inhabilitar la selección mientras se llena la misma, evitando así que se produzca inicialmente su evento SelectedIndexChanged.

La lista fija ya muestra el nombre de las asignaturas en las que se ha matriculado el alumno seleccionado (eso es lo que indica su propiedad DisplayMember) y, además, tenemos acceso a las notas correspondientes (eso es lo que indica su propiedad

ValueMember.) Cuando el usuario seleccione una de las asignaturas mostradas, se tiene que visualizar la nota correspondiente a esa asignatura. La acción de seleccionar un elemento en la lista genera el evento SelectedIndexChanged. Lo que tenemos que hacer es escribir el controlador para este evento:

```
        private void listaAsignatuasNotas_SelectedIndexChanged(object sender,
        EventArgs e)
        {
        if (listaAsignaturasNotas.SelectedIndex < 0) return;
        MessageBox.Show(listaAsignaturasNotas.SelectedValue.ToString());

}
```

La aplicación está finalizada. La ventaja que nos proporciona LINQ es que utilizando LINQ to SQL hemos sido capaces de acceder a una base de datos sin necesidad de tener que conocer los proveedores de datos de ADO.NET y el lenguaje de consultas SQL.

Contextos de corta duración

En la aplicación que hemos realizado utilizamos un contexto de larga duración; al ser un atributo de la clase *Form1* durará el tiempo que la aplicación se esté ejecutando, lo que puede suponer un abuso de recursos difícil de soportar en muchas ocasiones. Por eso, puede ser conveniente utilizar contextos de corta duración. La forma más usual de añadir un contexto de corta duración es mediante una sentencia using. Por ejemplo:

```
        using ( ModObjBDNotasAlumsDataContext contextoDeDatos =
                new ModObjBDNotasAlumsDataContext ())
        {
                // …
        }
```

REALIZAR CAMBIOS EN LOS DATOS

Los cambios que se efectúen en los objetos que forman la base de datos orientada a objetos virtualimagen del modelo de datos de una base de datos relacional, no se aplican a esta última hasta que se llame explícitamente al método SubmitChanges de DataContext.

Al llamar a SubmitChanges se desencadena el siguiente proceso:

1. LINQ to SQL examina el conjunto de objetos para determinar si ha habido cambios. Si hubo cambios, los objetos con cambios que dependen de otros se ordenan según sus dependencias.
2. LINQ to SQL inicia una transacción para encapsular la serie de ordenes individuales.
3. Los cambios en los objetos se convierten uno a uno en sentencias SQL y se envían al servidor. En este punto, cualquier error detectado por el gestor de la base de datos hace que: se detenga el proceso de envío, se inicie una excepción y se reviertan todos los cambios de la base de datos, como si no se hubiese enviado nada. Además, como DataContext mantiene un registro completo de todos los cambios, se puede intentar corregir el problema y llamar de nuevo a SubmitChanges. Para tener información de los objetos modificados, podemos invocar al método GetChangeSet.

```
       using (ModObjBDNotasAlumsDataContext contextoDeDatos =
                 new ModObjBDNotasAlumsDataContext())
            {
                // Cambios
                // ...
string sCambios = contextoDeDatos.GetChangeSet().ToString();
                MessageBox.Show(sCambios);
// ...

                // Enviar los cambios a la base de datos
try
                {
                    contextoDeDatos.SubmitChanges();
                }
                catch (Exception ex)
{
MessageBox.Show(ex.Message);
                    // Alternativa: hacer ajustes e intentarlo otra vez
                    // contextoDeDatos.SubmitChanges();
                }
            }
```

Cuando no se han realizado cambios, los resultados devueltos por GetChangeSet aparecen de la siguiente manera: *{Inserts: 0, Deletes: 0, Updates:0}*.

El método SubmitChanges sin argumentos ejecuta las sentencias adecuadas para intentar implementar los cambios en la base de datos. Si ocurre un error se procede como se ha indicado anteriormente en el punto 3. Hay otra versión, con un argumento (FailOnFirstConflict o ContinueOnConflict) que permite especificar la acción que se va a emprender si se produce un error en el envío:

contextoDeDatos.SubmitChanges(ConflictMode.FailOnFirstConflict);

El valor *FailOnFirstConflict* especifica que el intento de actualizar la base de datos se detendrá en cuando se detecte el primer conflicto y el valor *ContinueOnConflict* especifica que se deberían probar todas las actualizaciones de la base de datos y que los conflictos de concurrencia o simultaneidad se deberían acumular y devolver al final del proceso.

A continuación se explicarácómo se implementan las modificaciones de filas de una tabla, las inserciones de filas y el borrado de las mismas. Utilizaremos contextos de corta duración para implementar cada una de estas operaciones.

Modificar filas en la base de datos

Se puede actualizar las filas de una base de datos modificando los valores de los miembros de los objetos asociados a la colección Table<*clase_entidad*> y enviando después los cambios a la base de datos.

Para actualizar una fila de la base de datos seguimos estos pasos:

1. Se realiza una consulta para obtener la fila a modificar.
2. Se ejecuta la consulta y se cambian los valores de las columnas implicadas.
3. Se envían los cambios a la base de datos.

El siguiente ejemplo, busca en la tabla alums_asigs la fila (objeto alum_asig) correspondiente al alumno y asignatura especificados y modifica la nota, guardando después los cambios en la base de datos.

```
       // Obtener los datos
int idAlum = 1234567;
int idAsig = 1;
       float notaAlum = 87.4F;

// Consulta para obtener la fila a modificar
       var consulta =
       from al_as in contextoDeDatos.alums_asigs
              where al_as.id_alumno == idAlum &&
       al_as.id_asignatura == idAsig
select al_as;
       if (consulta.Count() == 0)
       {
       MessageBox.Show("La consulta no tiene elementos");
       return;
       }

       // Ejecutar la consulta y cambiar los valores de las columnas implicadas
       foreach (alum_asig al_as in consulta)
al_as.nota = notaAlum;

       // Enviar los cambios a la base de datos
try
       {
       contextoDeDatos.SubmitChanges();
       MessageBox.Show("Cambios realizados");
       }
       catch (Exception ex)
       {
       MessageBox.Show(ex.Message);
}
```

Insertar filas en la base de datos

Para insertar filas en una base de datos, se agregan objetos a la colección Table<*clase_entidad*> correspondiente y después se envían los cambios a la base de datos.

Para insertar una fila en la base de datos seguimos estos pasos:

1. Se crea un nuevo objeto que incluya los datos de esa fila.
2. Se agrega el nuevo objeto a la colección Table que está asociada a la tabla de destino en la base de datos. Para ello, Table facilita el método InsertOnSubmit(*entidad*). Los nuevos objetos serán registrados en el ObjectStateManager con el estado Added.
3. Se envía el cambio a la base de datos.

El siguiente ejemplo crea un nuevo objeto de tipo alumno, se llena con los valores adecuados y se agrega a la colección alumnos. Finalmente, se envía el cambio a la base de datos.

```
       // Obtener los datos
int idAlum = 2233434;
       string nomAlum = "Juan Lopez Perez";
```

```
// Crear un nuevo objeto alumno.
       alumno alumn = new alumno
       {
       id_alumno = idAlum,
       nombre = nomAlum
};

       // Añadir a la colección alumnos.
       contextoDeDatos.alumnos.InsertOnSubmit(alumn);

       // Enviar los cambios a la base de datos.
try
       {
       contextoDeDatos.SubmitChanges();
       MessageBox.Show("Cambios realizados");
}
       catch (Exception ex)
       {
             MessageBox.Show(ex.Message);
}
```

Si la operación de inserción no tiene sentido, por ejemplo, porque intentamos insertar una fila en la tabla *alumnos* y el identificador del alumno ya existe en la tabla *alumnos*, se elevará la excepción correspondiente.

Para añadir un objeto alums_asigs, se crea un nuevo objeto relacionado con otro objeto del contexto de objetos. Para añadirlo procedemos de alguna de las dos formas siguientes:

- Para relaciones de uno o varios, llamamos al método Add de EntitySet<*clase_entidad*> y especificamos el objeto relacionado.
- Para relaciones de varios a uno, establecemos la propiedad *clase_entidad* correspondiente al atributo de tipo EntityRef<*clase_entidad*> con el objeto relacionado.

El siguiente ejemplo crea un nuevo objeto de tipo *alum_asig*, se asignan los valores adecuados, se asignan las relaciones de este objeto con los objetos *alumno y asignatura* respectivos y se agrega a la colección *alums_asigs*. Finalmente, se envía el cambio a la base de datos.

```
       // Obtener los datos
       int idAlum = 2233434;
       int idAsig = 2;

       // Consulta para obtener el alumno a matricular
       var consulta1 = from alum in contextoDeDatos.alumnos
             where alum.id_alumno == idAlum
              select alum;

      // Consulta para obtener la asignatura de la que se va a matricular
var consulta2 = from asig in contextoDeDatos.asignaturas
where asig.id_asignaturas == idAsig
             select asig;

      if (consulta1.Count() == 0 || consulta2.Count() == 0)
{
```

```
MessageBox.Show("El alumno o la asignatura no existen");
            return;
      }
      alumno alumno = consulta1.First();
      asignatura asignatura = consulta2.First();

      // Crear el objeto alum_asig secundario de alum.
alum_asig alumAsig = new alum_asig
{
            id_alumno = idAlum,
            id_asignatura = idAsig,
            nota = 78.2F
      };

      // Establecer las relaciones
      alumAsig.alumno = alumno;          // N:1
      alumAsig.asignatura = asignatura;  // N:1
      // O bien:
      //alumno.alums_asigs.Add(alumAsig);       // 1:N
      //asignatura.alums_asigs.Add(alumAsig);   // 1:N

      // Añadirlo  a la coleccion alums_asigs.
      contextoDeDatos.alums_asigs.InsertOnSubmit(alumAsig);
      // O bien alumno.alums_asigs.Add(alumAsig);

      // Enviar los cambios a la base de datos
try
      {
            contextoDeDatos.SubmitChanges();
       MessageBox.Show("Cambios realizados");
}
      catch (Exception ex)
      {
            MessageBox.Show(ex.Message);
}
```

Borrar filas en la base de datos

Para eliminar filas de una base de datos, se quitan los objetos deseados de la colección Table<*clase_entidad*> correspondiente y después se envían los cambios a la base de datos.

LINQ to SQL no admite operaciones de eliminación en cascada. Siqueremos eliminar una fila de una tabla que tiene restricciones, debemos realizar una de las siguientes tareas:

- Se escribe código para eliminar primero los objetos secundarios que impiden que se elimine el objeto primario.
- Se establece la regla ON DELETE CASCADE en la restricción FOREIGN KEY de la base de datos.

De acuerdo con el primer punto anterior, para eliminar una fila de la base de datos seguimos estos pasos:

1. Se Realiza una consulta para obtener los objetos secundarios que impiden que se elimine el objeto primario.
2. Se ejecuta la consulta para eliminar los objetos secundarios. Para ello, la clase Table proporciona el método DeleteOnSubmit (*entidad*) que permite marcar el objeto para borrarlo (será registrado en el ObjectStateManager con el estado *Deleted*). La fila correspondiente será eliminada del origen de datos cuando se ejecute SubmitChanges.
3. Se Realiza una consulta para obtener el objeto primario a eliminar.
4. Se ejecuta la consulta para eliminar el objeto primario.
5. Se envían los cambios a la base de datos.

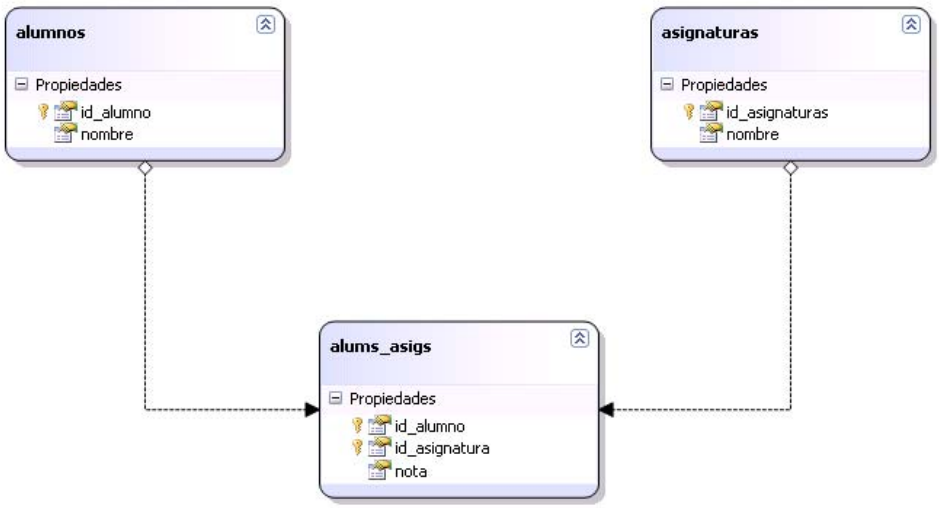
El siguiente ejemplo, elimina el alumno especificado. Para ello, busca en la tabla *alums_asigs* las filas correspondientes al alumno (objetos *alum_asig*) y después elimina estos objetos y su objeto primario (de tipo *alumno*), guardando a continuación los cambios en la base de datos.

```
       // Obtener los datos
int idAlum = 1234567;

      // Consulta para obtener los objetos secundarios de idAlum
var consulta1 =
      from alum in contextoDeDatos.alums_asigs
       where alum.id_alumno == idAlum
       select alum;

       if (consulta1.Count() != 0)
{
       foreach (var al_as in consulta1)
{
       // Borrar los objetos secundarios
                    contextoDeDatos.alums_asigs.DeleteOnSubmit(al_as);
              }
       }

       // Consulta para obtener el objeto primario idAlum
       var consulta2 =
       from alum in contextoDeDatos.alumnos
       where alum.id_alumno == idAlum
       select alum;

       if (consulta2.Count() != 0)
{
       foreach (var alum in consulta2)
{
       // Borrar el objeto primario
                    contextoDeDatos.alumnos.DeleteOnSubmit(alum);
}
}

       if (consulta1.Count() == 0 && consulta2.Count() == 0)
{
       MessageBox.Show("La consulta no tiene elementos");
              return;
       }
```

```
        // Enviar los cambios a la base de datos
try
        {
contextoDeDatos.SubmitChanges();
        MessageBox.Show("Cambios realizados");
        }
        catch (Exception ex)
        {
                MessageBox.Show(ex.Message);
}
```

PRUEBAS DE DESEMPEÑO

Las siguientes pruebas se realizaron utilizando una base de datos creada en SQL Server 2008, la cual está compuesta de la siguiente manera:

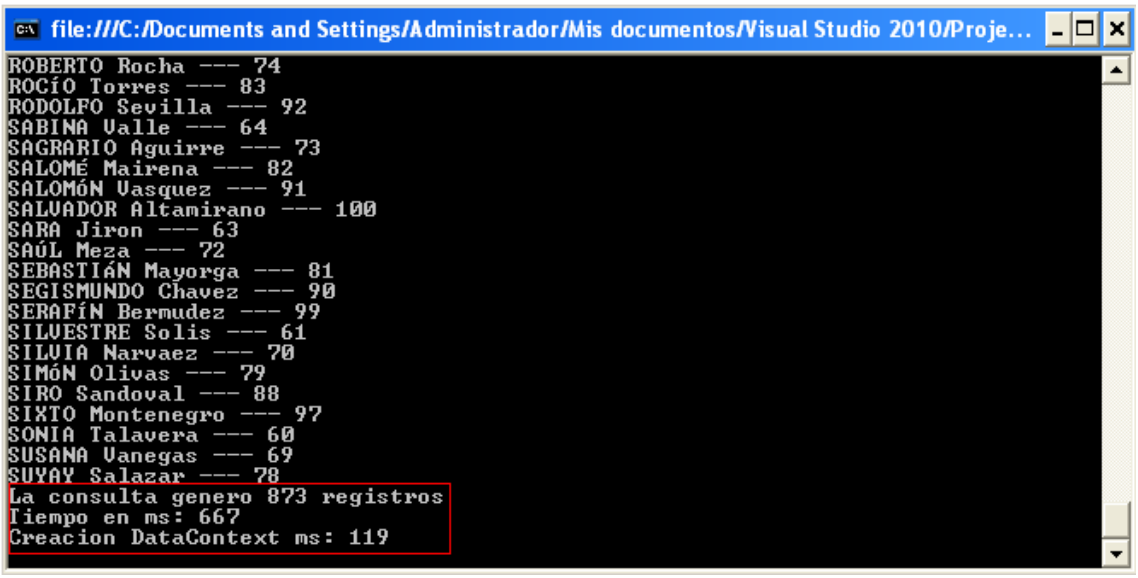


La tabla alumnos contiene 2500 registros, la tabla alums_asigs contiene 10000 registros y la tabla asignaturas contiene 4 registros.

Consulta 1: Obtener los nombres y las notas de los alumnos que aprobaron la clase programación visual

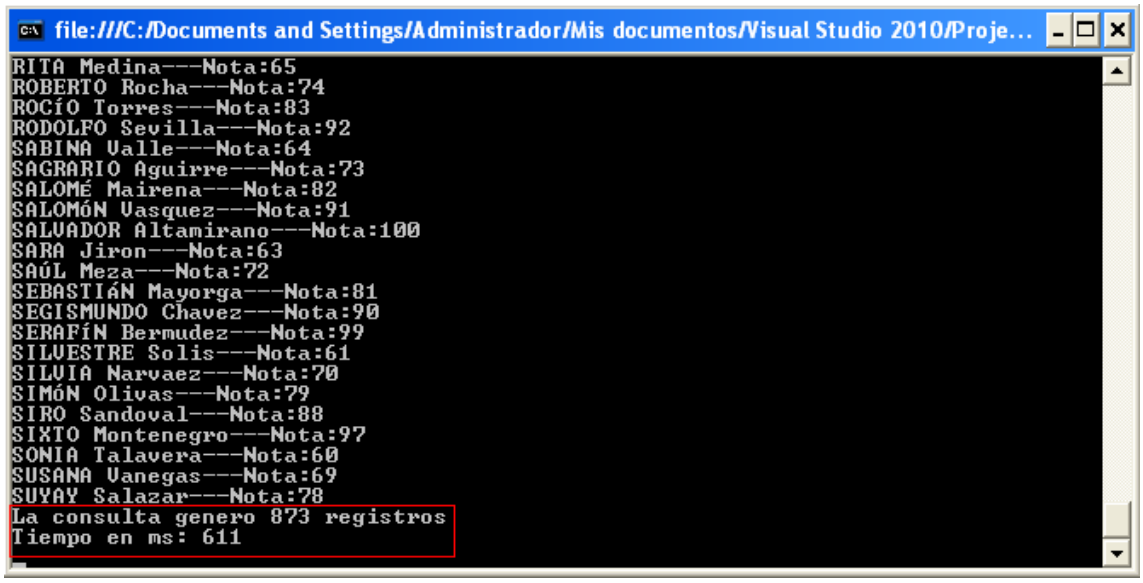
Consulta realizada utilizando LINQ

```
var alums = from alum in contextoDeDatos.alum_asig
       where alum.nota >= 60 && alum.id_asignatura == 1
       select new { Nombre = alum.alumno.nombre, Nota = alum.nota };
```



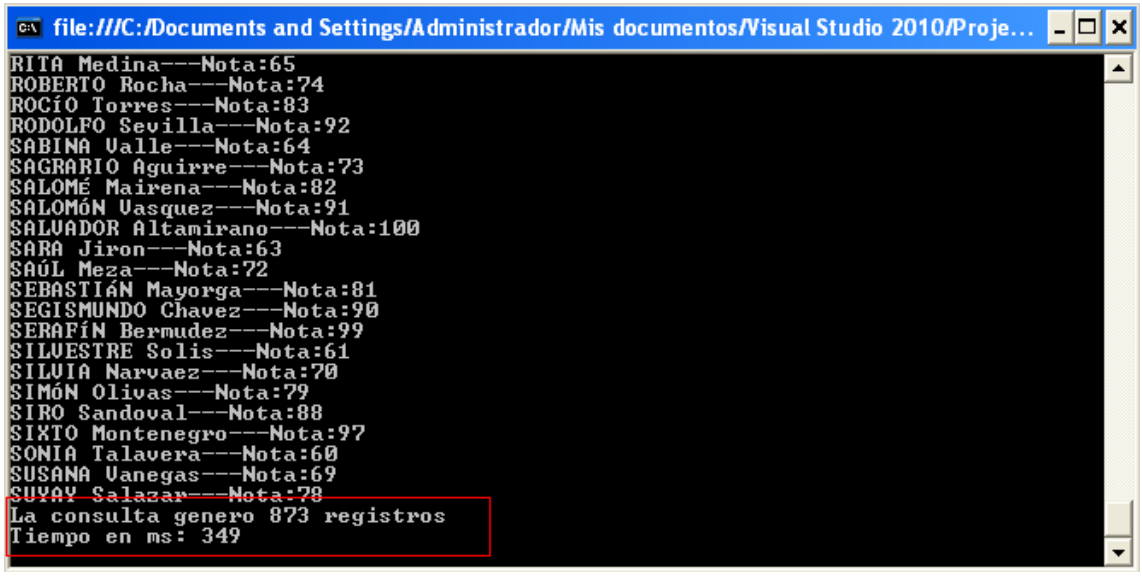
Consulta SQL con Data Reader de ADO.NET

```
string query = "select alumnos.nombre, alums_asigs.nota from alumnos inner join
alums_asigs on alumnos.id_alumno = alums_asigs.id_alumno where id_asignatura = 1
and nota >= 60";
```



Consulta SQL con Data Reader de ADO.NET utilizando vista.

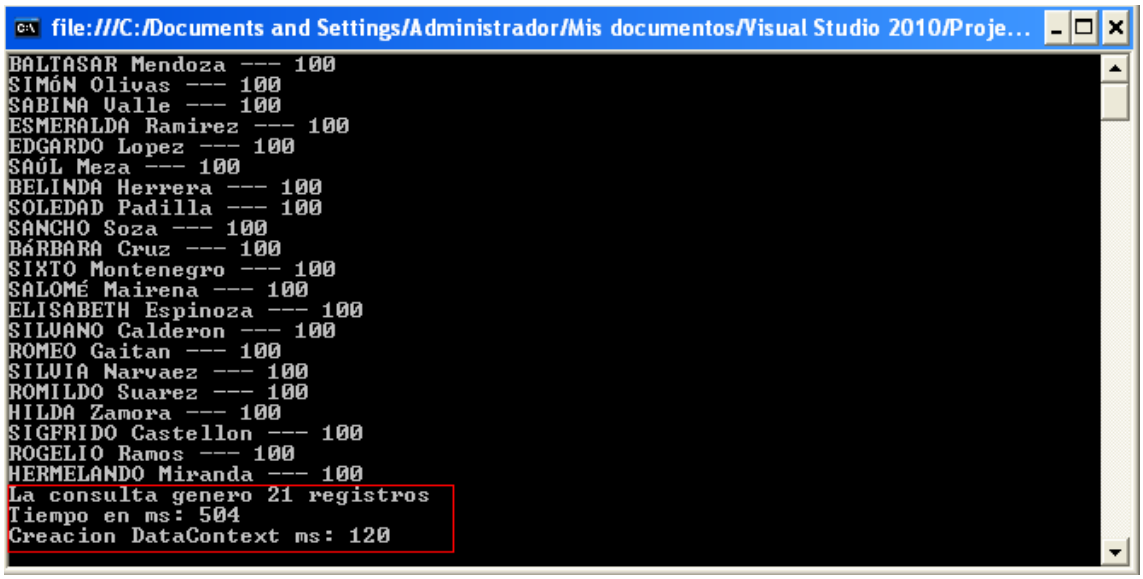
```
string query = "select * from vista_consulta1";
```



Consulta 2:Obtener los nombres de los alumnos que tienen 100 en la asignaturaFísica I

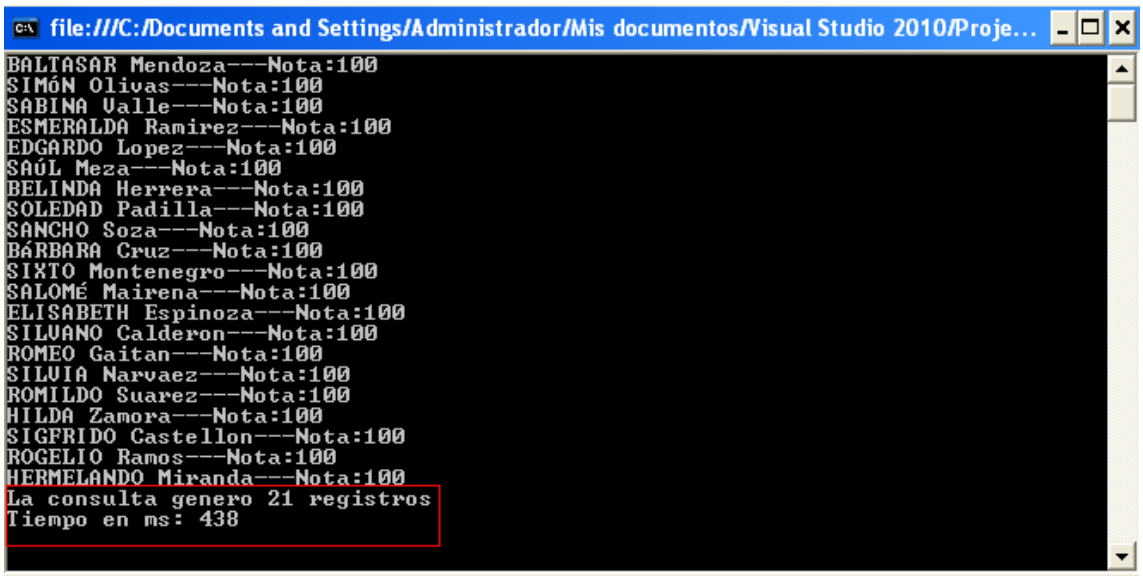
Consulta realizada utilizando LINQ

```
var alums = from alum in contextoDeDatos.alum_asig
       where alum.nota == 100 && alum.id_asignatura == 2
       select new { Nombre = alum.alumno.nombre, Nota = alum.nota };
```



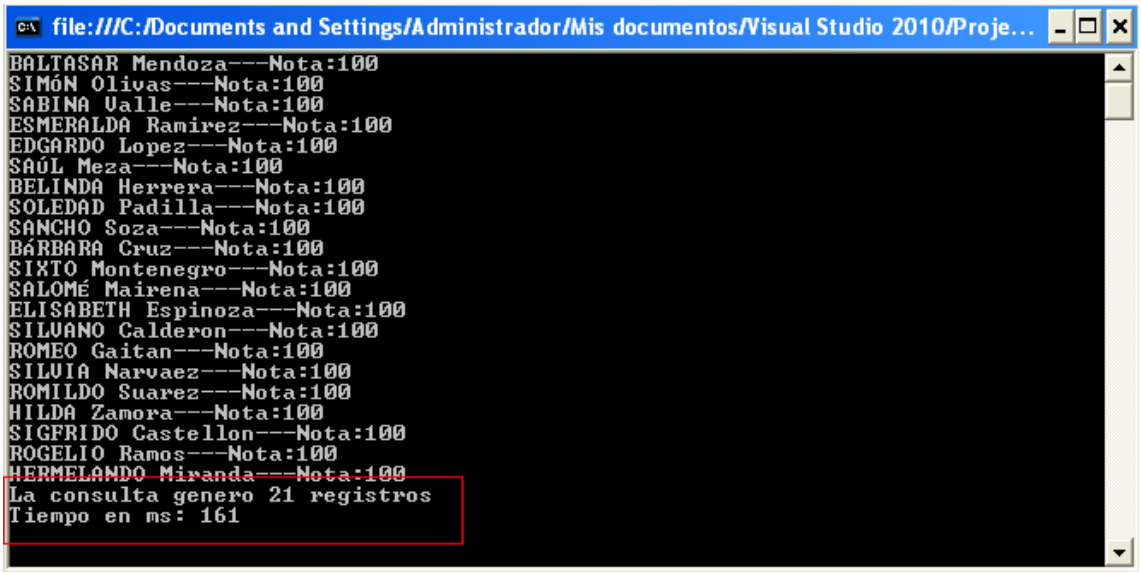
Consulta SQL con Data Reader de ADO.NET

```
string query = "select alumnos.nombre from alumnos inner join alums_asigs on
alumnos.id_alumno = alums_asigs.id_alumno where id_asignatura = 2 and nota =
100";
```



Consulta SQL con Data Reader de ADO.NET utilizando vista.

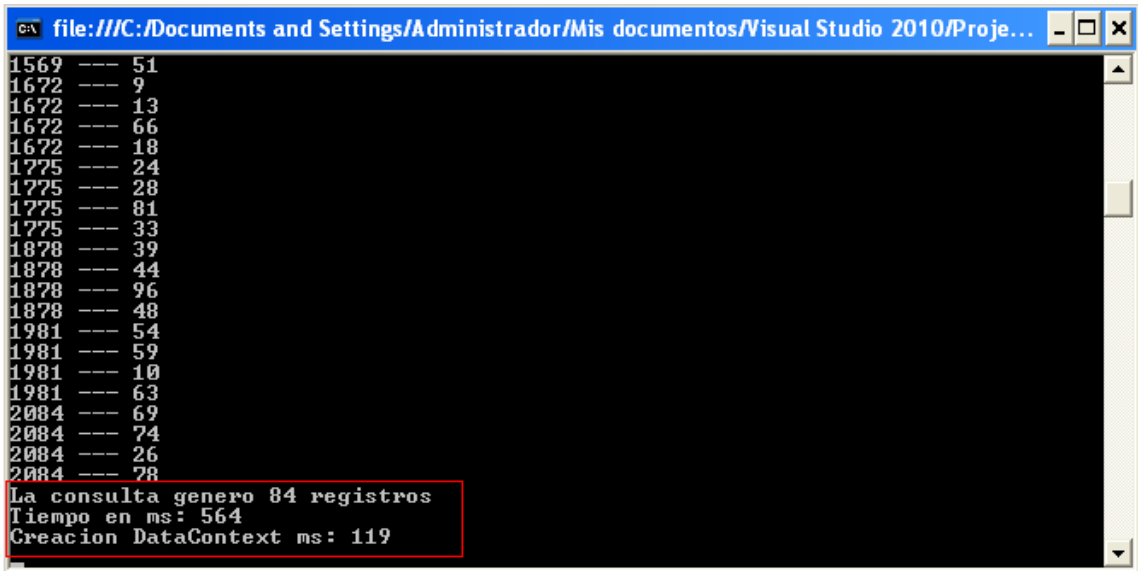
```
string query = "select * from vista_consulta2";
```



Consulta 3: obtener el identificador de cada alumno y las notas de los estudiantes que se llamen "BENITO Jarquin"

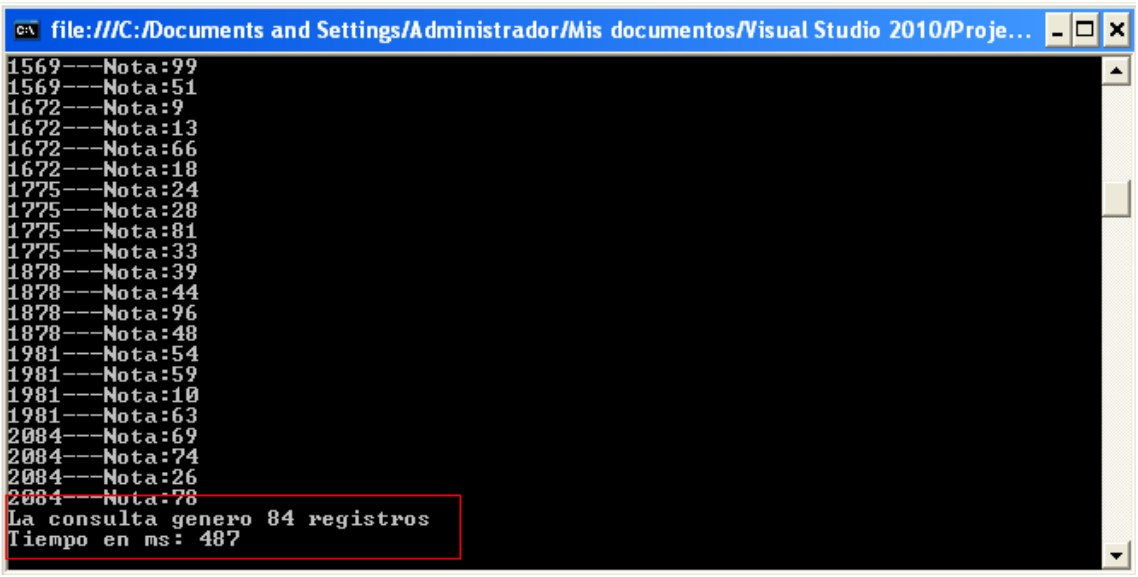
Consulta realizada utilizando LINQ

```
var alums = from alum in contextoDeDatos.alum_asig
       where alum.alumno.nombre == "BENITO Jarquin"
       select new { id = alum.id_alumno, Nota = alum.nota };
```



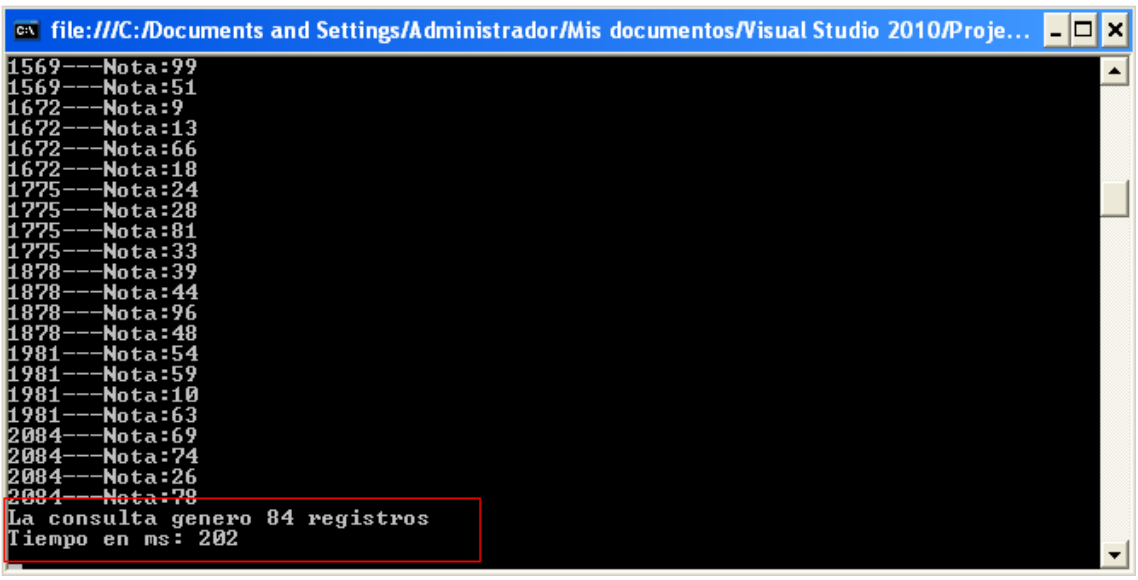
Consulta SQL con Data Reader de ADO.NET

```
string query = "select alums_asigs.id_alumno, alums_asigs.nota from alums_asigs
inner join alumnos on alumnos.id_alumno = alums_asigs.id_alumno where nombre like
'BENITO Jarquin'";
```



Consulta SQL con Data Reader de ADO.NET utilizando vista.

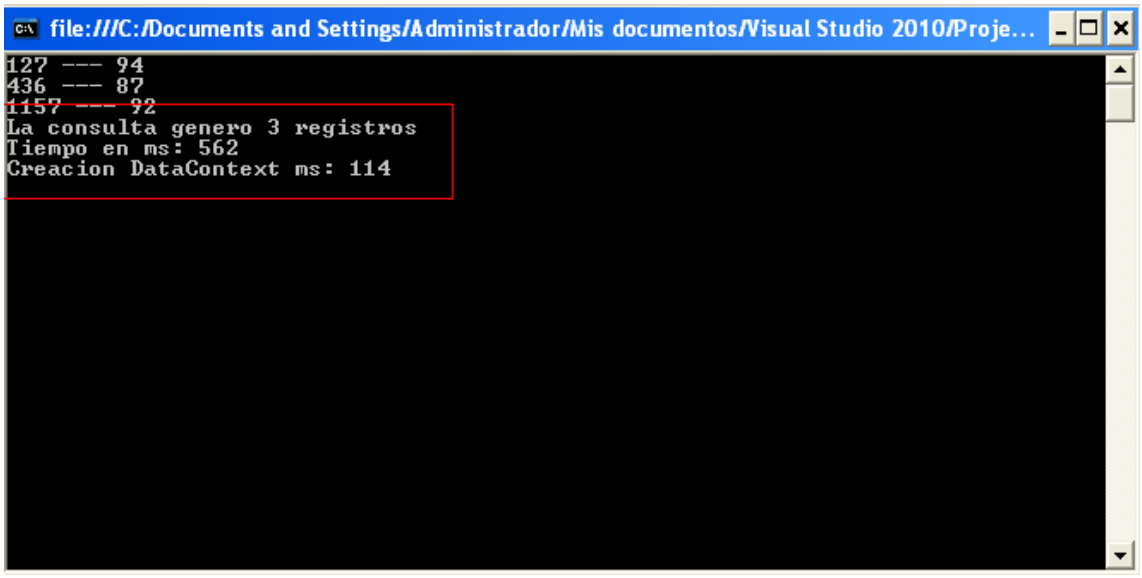
```
string query = "select * from vista_consulta3";
```



Consulta 4: Obtener el identificador de cada alumno y las notas de los estudiantes que se llamen BENITO Jarquin y que hayan obtenido notas mayores que 80 en la asignatura Comunicación de datos.

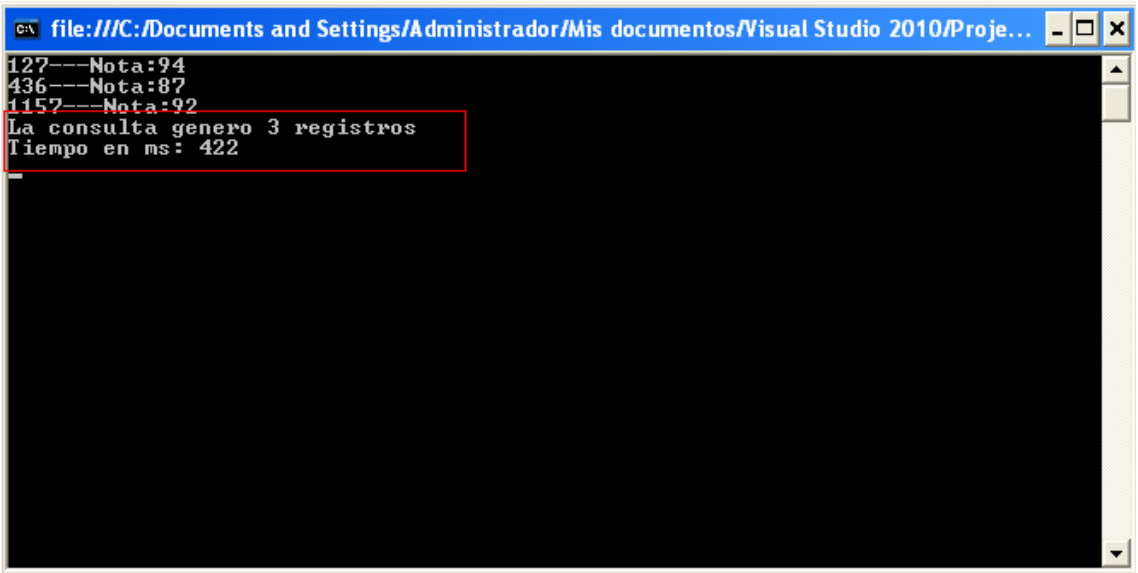
Consulta realizada utilizando LINQ

```
var alums = from alum in contextoDeDatos.alum_asig
       where alum.alumno.nombre == "BENITO Jarquin" && alum.id_asignatura == 4 &&
       alum.nota > 80
        select new { id = alum.id_alumno, Nota = alum.nota };
```



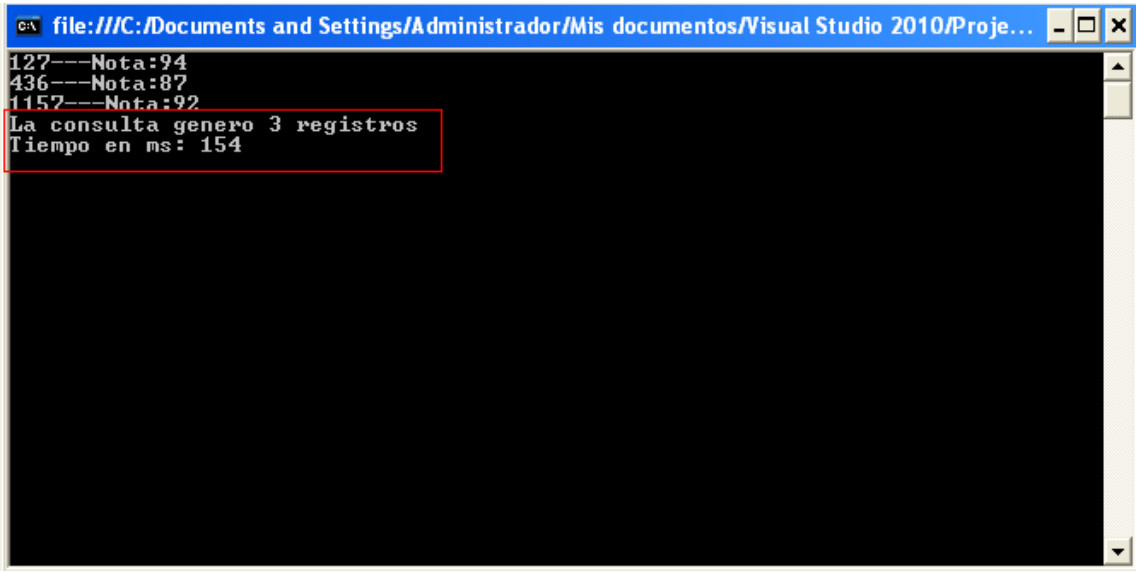
Consulta SQL con Data Reader de ADO.NET

```
string query = "select alums_asigs.id_alumno, alums_asigs.nota from alums_asigs
inner join alumnos on alumnos.id_alumno = alums_asigs.id_alumno where nombre like
'BENITO Jarquin' and nota>80 and id_asignatura = 4";
```



Consulta SQL con Data Reader de ADO.NET utilizando vista.

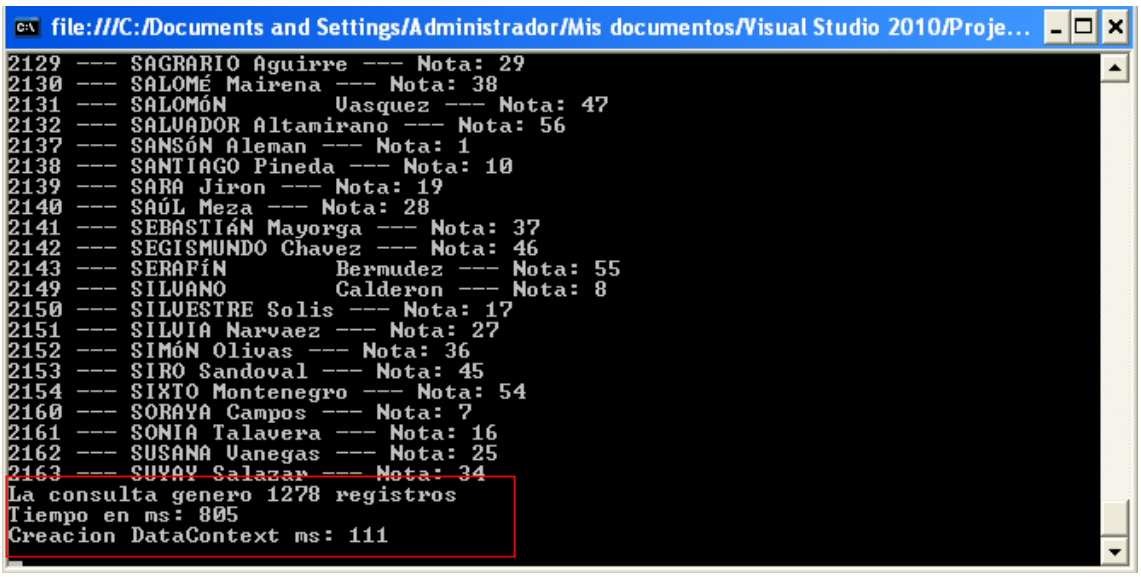
```
string query = "select * from vista_consulta4";
```



Consulta 5: Obtener el identificador de cada alumno, los nombres y las notas de todos los estudiantes reprobados en la asignatura Cálculo II.

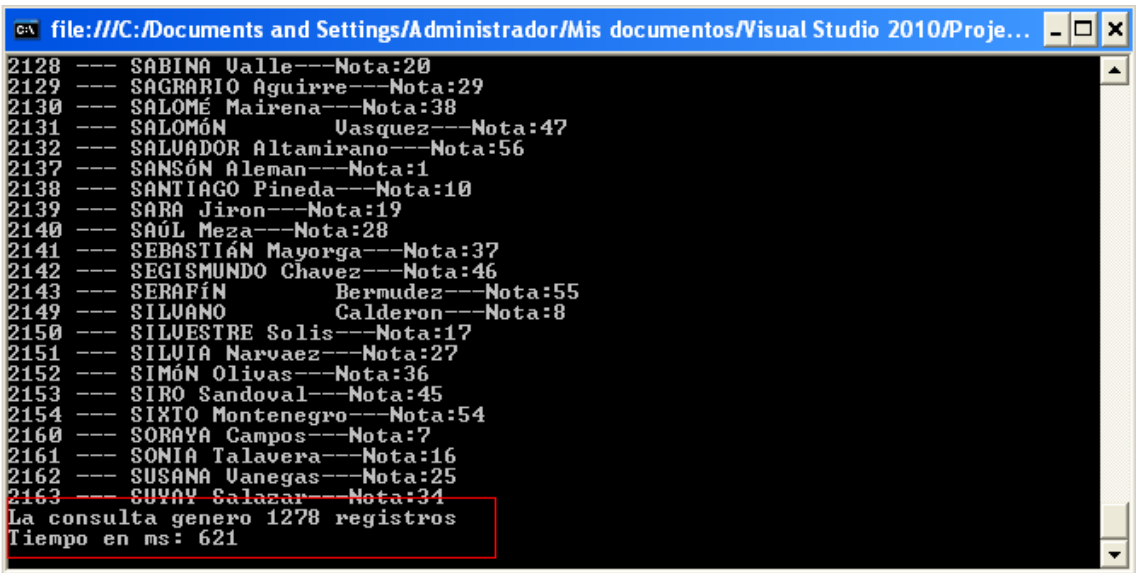
Consulta realizada utilizando LINQ

```
var alums = from alum in contextoDeDatos.alum_asig
       where alum.nota < 60 && alum.id_asignatura == 3
       select new { id = alum.id_alumno, Nota = alum.nota, Nombre =
       alum.alumno.nombre };
```



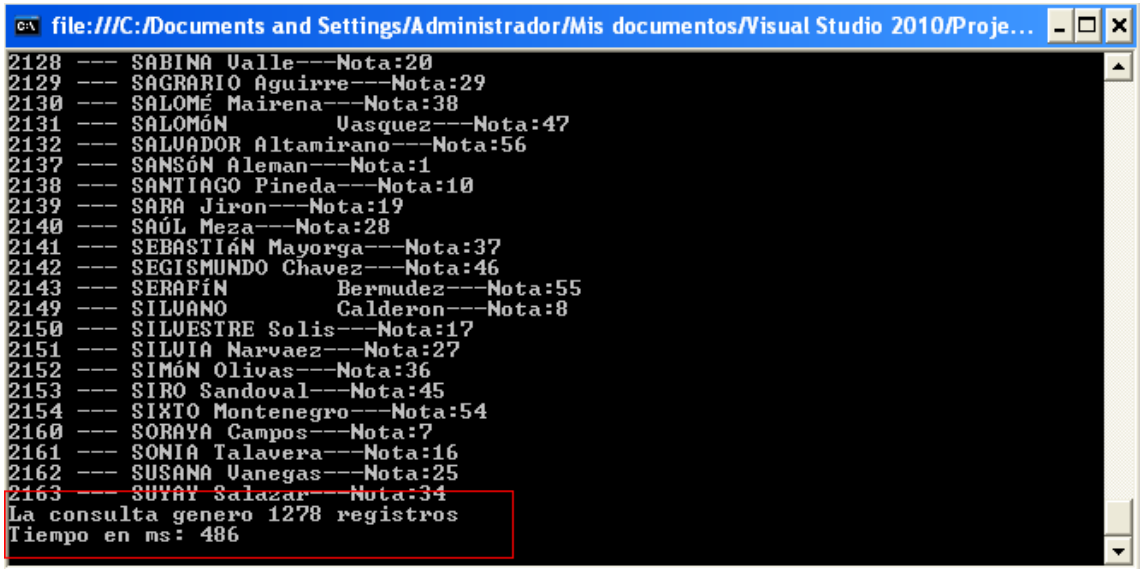
Consulta SQL con Data Reader de ADO.NET

```
string query = "select alumnos.id_alumno, alumnos.nombre, alums_asigs.nota from
alumnos inner join alums_asigs on alumnos.id_alumno = alums_asigs.id_alumno where
id_asignatura = 3 and nota < 60";
```



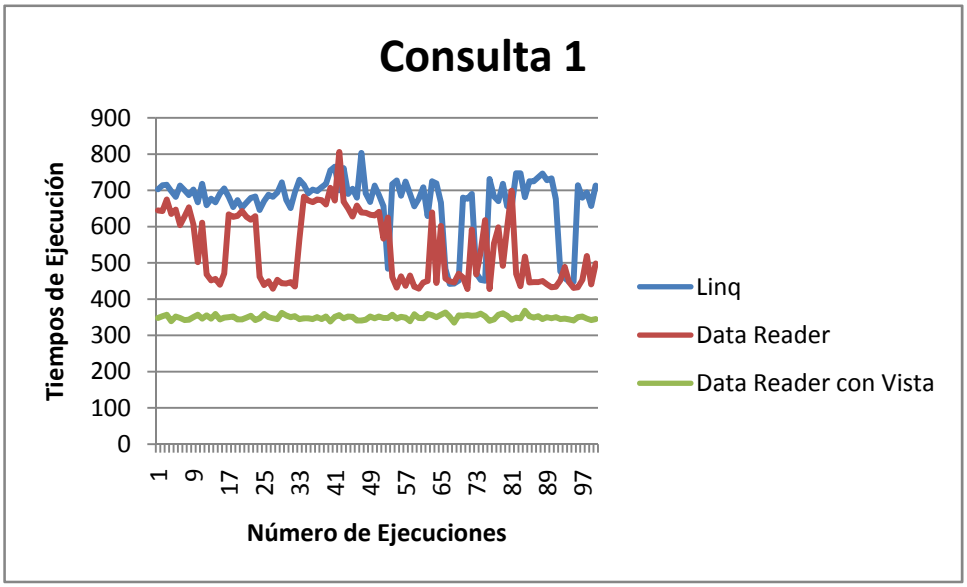
Consulta SQL con Data Reader de ADO.NET utilizando vista.

```
string query = "select * from vista_consulta5";
```

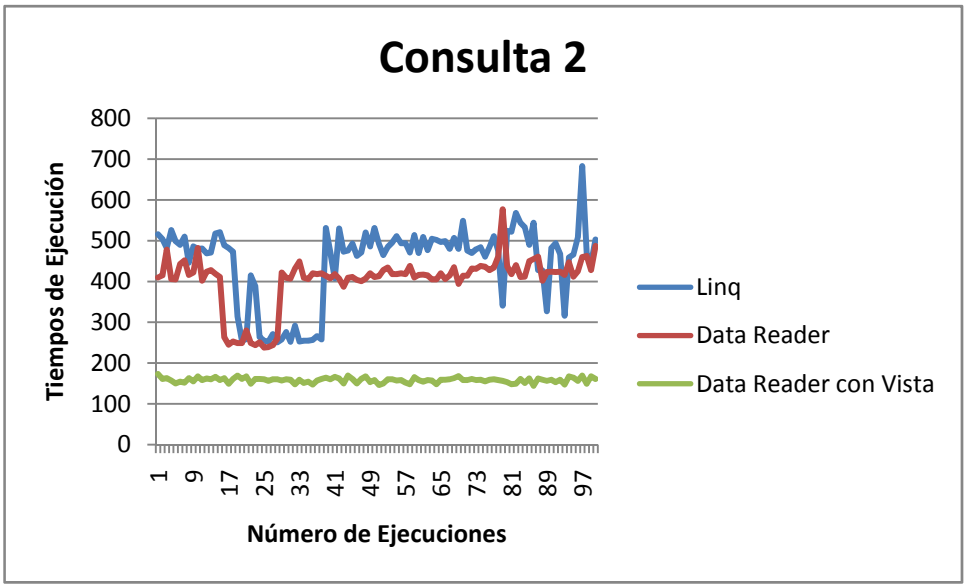


RESULTADOS

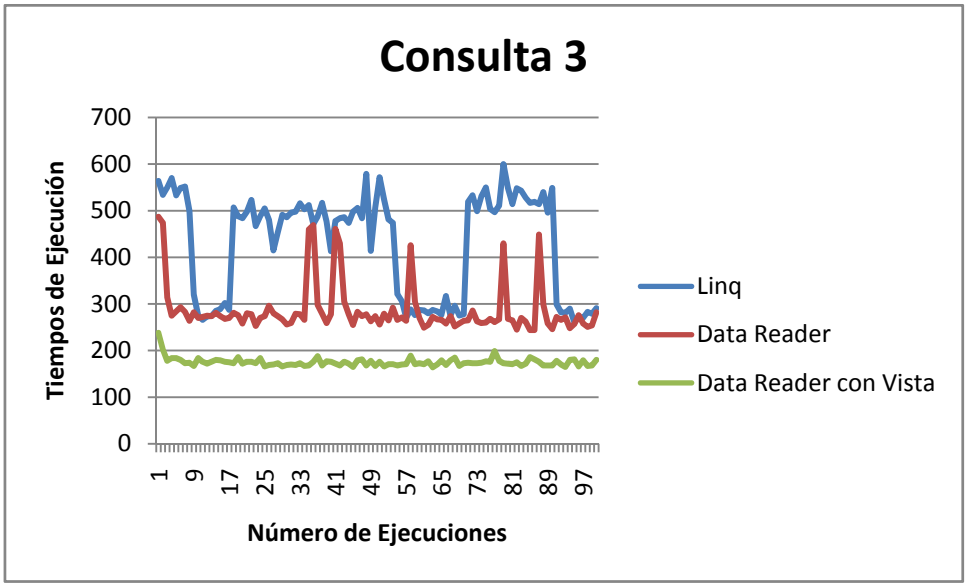
Consulta 1: Obtener los nombres y las notas de los alumnos que aprobaron la clase programación visual.



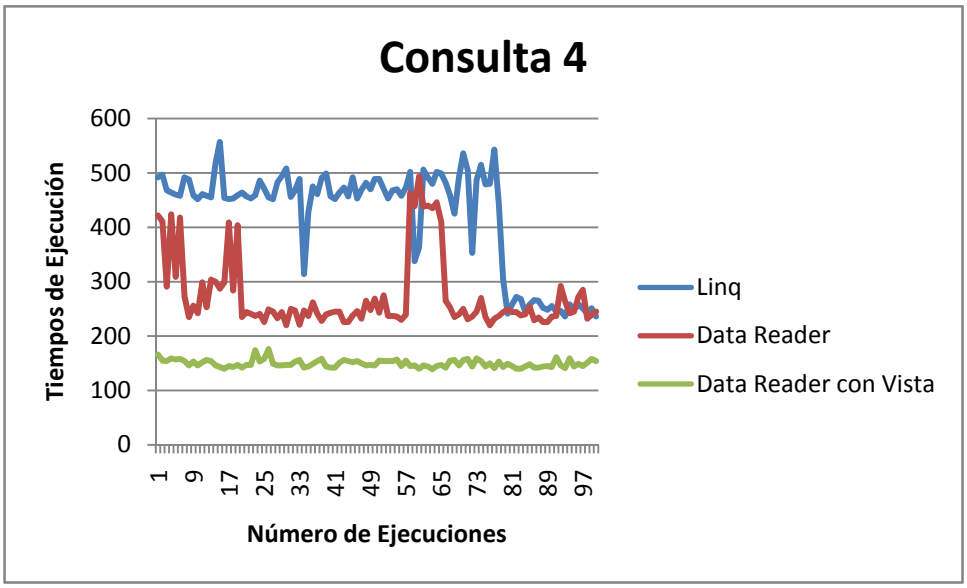
Consulta 2: Obtener los nombres de los alumnos que tienen 100 en la asignatura Física.



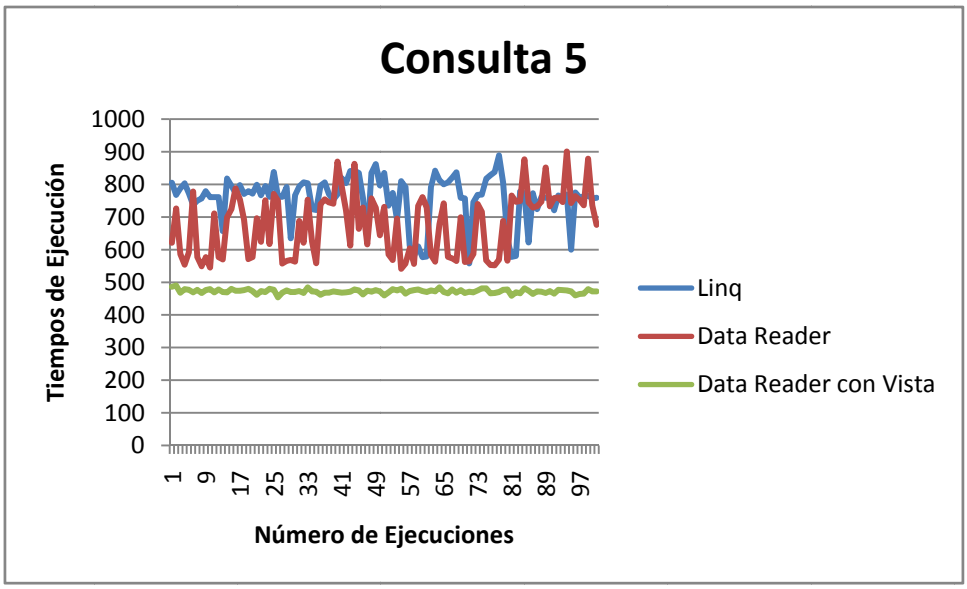
Consulta 3: obtener el identificador de cada alumno y las notas de los estudiantes que se llamen “BENITO Jarquin”.



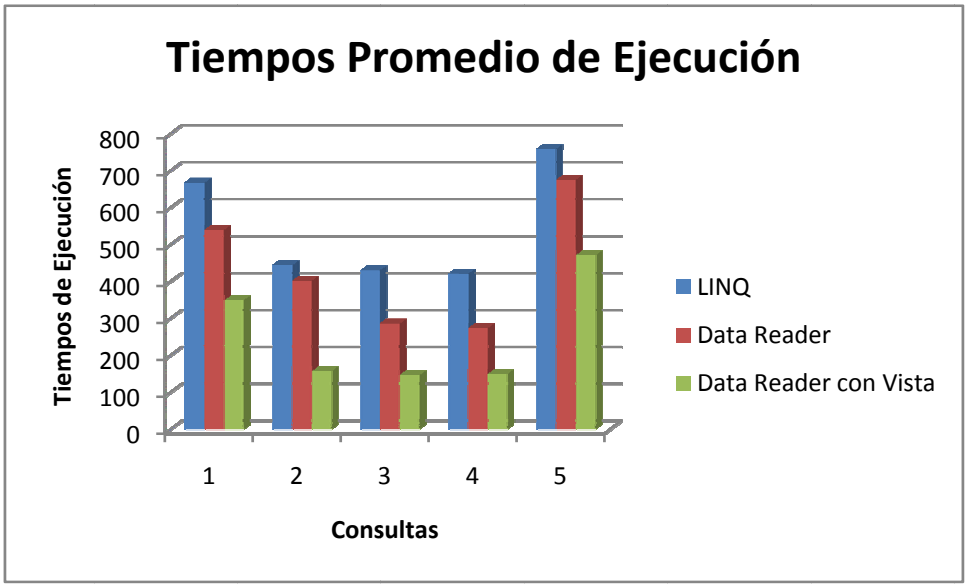
Consulta 4: Obtener el identificador de cada alumno y las notas de los estudiantes que se llamen BENITO Jarquin y que hayan obtenido notas mayores que 80 en la asignatura Comunicación de datos.



Consulta 5: Obtener el identificador de cada alumno, los nombres y las notas de todos los estudiantes reprobados en la asignatura Cálculo II.



Tiempos Promedio de Ejecución			
Consulta	**LINQ**	**Data Reader**	**Data Reader con Vista**
1	668,41	539,86	349,63
2	445,19	402,75	158,08
3	430,31	286,64	147,33
4	421,84	274,2	149,63
5	758,81	676,09	472,24



CONCLUSIONES

Con la realización de este trabajo hemos cumplido con los objetivos planteados. En cuanto a la investigación del Lenguaje de Consultas Integrado LINQ se logro entender las características que este posee, además de las facilidades que nos proporciona.

Una vez estudiado el lenguaje LINQ fuimos capaces de realizar pruebas de desempeño para luego establecer las ventajas y desventajas que nos proporciona cada una de las tecnologías.

Hemos comprobado a través de las pruebas de desempeño realizadas que LINQ nos proporciona mayor flexibilidad al momento de realizar las consultas, debido a que se trata de un lenguaje orientado a objetos,además nos facilita el acceso a los datos, con una consulta sencilla. Con respecto al tiempo de ejecución concluimos que LINQ no es tan potente como ADO.NET utilizando Data Reader.

RECOMENDACIONES

Como recomendaciones invitamos a seguir con el estudio del lenguaje LINQen las versiones posteriores, ya que la evolución de este lenguaje ha demostrado mejoras significativas entre cada versión.

Instamos a los programadores y desarrolladores de aplicaciones a seguir utilizando el lenguaje SQL, ya que mediante las pruebas realizadas se comprobó que ofrece mejor rendimiento en cuanto a tiempos de respuesta en comparación con LINQ.

REFERENCIAS

- Fundamentos de Bases de Datos. Silberschatz, Korth, Sudarshan. Quinta edición.
- Enciclopedia de Microsoft Visual C#. Francisco Javier Ceballos. Tercera edición.
- http://msdn.microsoft.com/es-es/library
- LINQ in Action. Fabrice Marguerie, Jim Wooley. Manning.