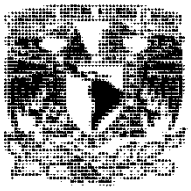


UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ARAGÓN

LINQ TO SQL COMO UNA EXTENSIÓN FUNCIONAL DE UN MODELO DE PROGRAMACIÓN QUE SIMPLIFICA Y UNIFICA LA IMPLEMENTACIÓN DE ACCESO A CUALQUIER TIPO DE DATOS.

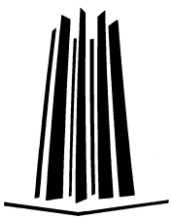
T E S I S

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

P R E S E N T A:

CRUZ JIMÉNEZ JOSÉ ANTONIO

Asesor: M. en C. Jesús Hernández Cabrera.



NEZAHUALCÓYOTL, ESTADO DE MÉXICO 2009

Agradecimientos.

Dedico este trabajo a mis padres:

José Delfino Cruz Cruz.

Juana Jiménez Baltazar.

Por su apoyo y cariño incondicional proporcionado antes y durante la realización de la tesis.

También se lo dedico a mis hermanos:

C. D. Edith.

Julio Cesar.

Lourdes.

Adriana.

Rosalba.

Javier.

Gracias.

ÍNDICE

INTRODUCCIÓN.

En el desarrollo de aplicaciones de software, cualquiera que sea el lenguaje de programación en el que un programador implemente la lógica del funcionamiento que cubra las necesidades requeridas, en algún momento se tendrá la necesidad de acceder a datos, en cuyo caso pueden estar en los archivos de un disco, las tablas en una base de datos relacional, o documentos *XML* procedentes de la Web o, a menudo, una combinación de todos éstos. De esta forma se hace imprescindible la manipulación de los datos como un requisito para cada proyecto de software, sobre el que un equipo desarrollador trabajará, por lo que las consultas se vuelven inherentes a los principales lenguajes de programación. Sin embargo, resulta obvio que se trata de dos paradigmas diferentes. En el modelo relacional se trabaja sobre relaciones, tuplas y conjuntos. El paradigma orientado a objetos se basa en objetos, sus atributos y relaciones entre objetos. Cuando se quiere hacer que los objetos se vuelvan persistentes usando para ello una base de datos relacional, se puede notar que existe una discordancia entre estos dos paradigmas. Esto conlleva a que la manipulación de los datos resulta de una tarea común para las aplicaciones de software, sin embargo esto último también resulta en un frente al que un programador se puede encontrar, y de acuerdo a esto, *.NET* y otras plataformas orientadas a objetos, proporcionan una amplia gama de herramientas para interactuar y trabajar con los datos.

De esta forma se puede encontrar que algunas de estas herramientas operan de forma separada con el lenguaje de programación y con ello aplican su propia definición para poder trabajar con los datos y con el mismo lenguaje dando como resultado una especialización de cada conjunto de herramientas, a veces vistos como *frameworks*, que responden a las diferentes necesidades de desarrollo. Sin embargo, se puede observar algo muy importante que no se había logrado hasta ahora: profundizar en la integración del lenguaje de programación y los datos que son manipulados.

Dentro de este contexto, es *LINQ* y sus tecnologías presentadas en este trabajo, las que entran para hacer frente a esta problemática y para lo que han sido diseñadas, dando como resultado una nueva forma de escribir el código basado en consultas sobre los datos. Observando lo anterior se tiene, *grosso modo,* que *LINQ* es una tecnología genérica para acceder a cualquier conjunto de datos que pueda verse como una secuencia o conjunto de secuencias de elementos de un cierto tipo: arreglos, colecciones, documentos *XML* o bases de datos relacionales, estos son algunos ejemplos de las posibles fuentes en *LINQ* para los que *.NET Framework* 3.5 ofrece soporte predefinido.

Sin embargo, el concepto que implementa Microsoft en su tecnología de *LINQ* no resulta novedoso, mas no así la forma en cómo lo implementa, puesto que rompe la brecha existente entre el lenguaje de programación y los datos, simplificando la consulta sobre los datos y mejorando, a través del agregado de nuevos elementos, a los lenguajes de programación de *.NET* como *Visual Basic .NET* y *C#* 3.0 y sus respectivos compiladores, convirtiéndose en una forma más fácil de consultar los datos, lo cual se verá reflejado a largo plazo en el mantenimiento de las aplicaciones de software. Esto es lo que lo diferencia de las demás herramientas existentes que se verán en la primera parte de esta tesis.

Por otra parte, una porción de la tecnología de *LINQ* que también presento en esta tesis, *LINQ To SQL*, además de haber sido diseñada como una nueva forma de escribir código, responde a las dificultades técnicas encontradas cuando se usan bases de datos con los lenguajes de programación. Pero esto puede hacer hincapié en generar una serie de ideas que llevan a una asociación de LINQ relativa a la sola ejecución de consultas sobre bases de datos relacionales, pero como se mencionó anteriormente, esto no es así, ya que solo es una fracción con la que cuenta *LINQ.*

En el primer capítulo se dará una visión general de *LINQ* para ayudar a identificar las razones para usarlo con lo cual se presentarán otros proveedores que implementan la misma idea para otros lenguajes. Además se mostrarán de forma concisa los fundamentos de la arquitectura e infraestructura *.NET Framework* que hacen posible la implementación de la Tecnología LINQ. Inherente a esto, se observarán las características propias de *.NET Framework* para dar soporte a las operaciones de consulta, características que se encuentran en las diversas versiones que la componen. Todo lo anterior será unido en la realización de una consulta integrada en el lenguaje inicial para dar una idea de lo que se hablará en los siguientes capítulos.

En el capítulo 2 se dará inicio al tema de *LINQ*. Cuáles son los ensamblados, espacios de nombres y la versión de *.NET Framework* en el que se encuentra a disposición. Se realizarán expresiones de consulta desde el propio lenguaje haciendo uso de sus propios operadores de consulta, así como también se abordarán las extensiones con las que se dispone de *LINQ*. Posteriormente se pasará a mostrar los diferentes proveedores que implementan la misma idea para otros *SMBD*.

Posteriormente, el capítulo 3 se centrará sobre el manejo de datos relacionales haciendo uso de *LINQ*, es decir, se observarán particularidades básicas de *LINQ to SQL*, como por ejemplo la realización de consultas sobre los datos y su manipulación sobre la base de datos, además de ejecutar y manipular los resultados de procedimientos almacenados. Con esto, se puede eliminar código de acceso repetitivo y trabajar de forma fluida con el *SMBD SQL Server* haciendo uso del poder de *LINQ*.

En el capítulo 4 se mostrarán características avanzadas de *LINQ to SQL* con las cuáles se puede personalizar las funciones que esta extensión tiene para manejar datos relacionales. Se podrá observar cómo el programador puede manipular y ampliar aún más las características de *LINQ to SQL* del capítulo anterior.

Finalmente en el 5to y último capítulo de este trabajo se pretende realizar una aplicación web que implemente algunas de las características vistas en *LINQ* y una de sus extensiones: *LINQ to SQL*. Se mostrará como *LINQ* puede hacer más versátil, simple e intuitivo el desarrollo de una aplicación tanto para escritorio como para la Web: en la primera estará basada en la administración de una base de datos y la segunda estará encarga de la venta de artículos por internet, que es muy común en estos días.

CAPÍTULO I. ANÁLISIS DE LOS COMPONENTES SUBYACENTES QUE PERMITEN LA IMPLEMENTACIÓN DE LINQ.

1.1 BREVE INTRODUCCIÓN.

En esta tesis se presentarán los proveedores que implementan la idea de realizar de forma nativa sentencias SQL en el lenguaje de programación (esto incluye operaciones de consulta). Con esto se puede observar que *LINQ* no solo es la aparición de una nueva tecnología, sino se trata también del resultado de una nueva ideología que se ha formado a través del tiempo para poder manipular los datos en los lenguajes de programación orientados a objetos (cualquiera que fuese este). En primera instancia se abordará de forma muy breve *LINQ*, para posteriormente pasar a los proveedores que existen para Java y mostrarlos de forma concisa con lo cuál se pretende dar una idea clara de la elección de *LINQ* sobre estos proveedores. Es muy importante mencionar desde un principio que el lenguaje de programación a usar junto con *LINQ* en los ejemplos a lo largo de esta tesis será *C#* (en este caso *C# 3.0* versión aparecida en *.NET Framework 3.5* y que le agrega más funcionalidades a este lenguaje de programación además de modificaciones a su compilador), esto para brindar una sintaxis similar a quienes estén familiarizados con los lenguajes de programación basados en *C* o *C++*, como es el caso de *Java* y con esto se logre un mejor entendimiento sobre los conceptos que se abordan además de ser por motivos de espacio, con esto no se pretende demeritar a el lenguaje *Visual Basic .NET* (*VB 9.0*) que también tiene soporte para la tecnología *LINQ*.

1.2 ¿QUÉ ES LINQ?.

LINQ (*Language Intergated Query*) es un conjunto de extensiones expuestos tanto para los lenguajes *C# 3.0* y *VB .NET* como para *.NET Framework*, lo cual proporciona una integración tanto en el desarrollo, como ejecución de consultas para objetos, base de datos y *XML*, esto con el objetivo de reducir a un nivel mínimo la complejidad en el desarrollo de software y ayudar a fomentar la productividad. Con la tecnología de *LINQ*, el programador puede desarrollar consultas que están orientados hacia el lenguaje *SQL* (*Structured Query Language*) en los lenguajes como lo son *C# 3.0* y *VB .NET* sin tener que usar otro lenguaje como lo es *SQL* ó *XQUERY* (un lenguaje de consulta para acceder a datos *XML*).

LINQ se compone de varios elementos: el primero de ellos es una serie de reglas que especifican como las consultas se encuentran definidas dentro de los ensamblados de *.NET Framework 3.5* y en cuyo caso resultan de gran importancia porque establecen las reglas para añadir soporte a cualquier lenguaje aplicable a *LINQ*, y con esto dar soporte para consultar cualquier objeto. El segundo de ellos basado en estas especificaciones, es que LINQ incluye una serie de extensiones tanto para los lenguajes *C# 3.0* y *Visual Basic .NET* como para sus compiladores, además de un conjunto de librerías que proporciona una integración de consultas para objetos, base de datos y datos *XML* usando la sintaxis del propio lenguaje. En la siguiente tabla se pueden apreciar tanto los ensamblados como los espacios de nombres provistos en *.NET Framework 3.5*:

.NET FrameWork 3.5		
Tecnología	**Ensamblado**	**Espacio de nombres**
LINQ to Objects	System.Core.dll	System.Linq
LINQ to XML	System.Xml.Linq.dll	System.Xml.Linq
LINQ to SQL	System.Data.Linq.dll	System.Data.Linq
LINQ to Entities	System.Data.Entity.dll	System.Data.Objects y otros

En una de las extensiones de *LINQ*, *LINQ To SQL* (*LINQ para SQL*) las consultas, operaciones que se realizan sobre los objetos son cambiadas hacia conceptos propios de los lenguajes de *.NET Framework* y ensamblados, dando como resultado entidades que resultan familiares al programador tales como objetos y clases. *LINQ To SQL* se encarga de lidiar con las diferencias existentes entre la capa de negocio y la capa de acceso a datos haciendo uso de sentencias *SQL* para la capa de acceso a datos y para el programador en clases, objetos y procedimientos almacenados encapsulados como métodos. Uno de los elementos importantes, sin dejar atrás otros que se mencionarán en los subsecuentes capítulos, es la interfaz genérica *IEnumerable<T>*, la cual expone al enumerador para permitir una iteración en una colección de un tipo especificado, en cuyo caso *T* es el tipo de los objetos que se van a enumerar. En lo que respecta a *LINQ*, la importancia de la interfaz *IEnumerable<T>* reside en que cualquier tipo de datos que implemente esta interfaz pueda ser usado como fuente para expresiones de consulta (una colecciones de datos a consultar), en este caso los arreglos y las colecciones genéricas de *.NET Framework 2.0* la implementan. Para esta parte introductoria de *LINQ* se mostrará el código completo de una aplicación de consola, la cuál solo muestra los archivos de un directorio para posteriormente ordenarlos de acuerdo a su tamaño en disco, esto con el fin de demostrar cómo se encuentra expresada una consulta *SQL* en el lenguaje con *LINQ* y cómo es soportado por el propio lenguaje (los conceptos se abordarán posteriormente).

C#

```
using System;
using System.Collections.Generic;
//espacio de nombre con el que se puede hacer uso de LINQ sin hacer uso de sus extensiones
using System.Linq;
//espacio de nombre para poder mostrar los datos através de la consola
using System.IO;
namespace Linq {
//Declaración de la clase
    class Linq {
        static void Main(string[] args){
//se obtiene la fuente de datos, es decir un arreglo que contenga objetos
//del tipo FileInfo(esta clase provee métodos de instancia para manipular
//archivos) através del método GetFiles() de la clase DirectoryInfo
            FileInfo[] objetosFile = new DirectoryInfo(@"C:\LINQ").GetFiles();
//se procede a realizar una consulta SQL, esto para fines demostrativos.
//Solamente se seleccionaran los objetos FileInfo de acuerdo a una condición
//y se ordenan de acuerdo a su tamaño de forma descendente
            var seleccionarTodos = from archivo in objetosFile
                        where archivo.Name.contains("a")
                        orderby archivo.Length descending
                        select archivo;
//se itera sobre cada uno de los elementos, esto gracias a que el valor devuelto
//implementa la interfaz IEnumerable<T>, en este caso seleccionarFiltrado
            foreach (var res in seleccionarFiltrado){
//Como los objetos son del tipo FileInfo, sus miembros publicos son accesibles
//en este caso se muestran el nombre,longitud y ruta
                Console.WriteLine("Nombre: " + res.Extension);
                Console.WriteLine("Longitud: " + res.Length);
                Console.WriteLine("Path: " + res.FullName);
                Console.WriteLine("\n");
            }
            Console.WriteLine("Total Archivos: " +
                                        seleccionarTodos.Count().ToString());
        }
    }
}
```

Como se puede observar, el código anterior resulta muy descriptivo, se muestra que *LINQ* resulta más fácil de utilizar en el lenguaje, sin embargo el compilador de *C# 3.0* convierte la consulta expresada anteriormente en llamadas a métodos de extensión (cuya definición se verá más adelante en el apartado *1.2.11.1 Métodos de extensión*) y libera al programador de realizar procesos complejos, centrándolo en escribir solamente el código de la lógica del negocio.

1.3 JoSQL

JoSQL (*SQL for Java Objects*) es una *API* que le permite a un programador realizar una consulta *SQL* sobre una colección de objetos a través del lenguaje de programación Java, con lo cuál se facilita la tarea de realizar operaciones como buscar, ordenar y agrupar cualquier objeto en Java y con lo que se logra un mejor beneficio en la obtención de datos sobre una colección de objetos con Java. Actualmente esta *API* se encuentra en su versión 2.0 cuya salida se realizó el 25/Sep/2008 al momento de escribir esta tesis. Para ello solo se debe tener acceso al archivo *JoSQL-2.0.jar* a través de la dirección *http://josql.sourceforge.net* y hacer referencia al paquete *org.josql.** . Por ejemplo, para poder obtener una colección de archivos .txt que hayan sido modificados en un periodo determinado (enero de 2009), se tendría el siguiente fragmento de código realizado en el lenguaje java con una sentencia *SQL* usando esta *API*:

Java

```
SELECT * FROM   java.io.File
WHERE  name LIKE '%.txt'
AND    lastModified BETWEEN toDate ('01-01-2009') AND toDate ('31-01-2009')
```

- ***Nota: Es conveniente mencionar desde un principio que esta sentencia es una cadena.***

JoSQL trata cada objeto de forma análoga a un "renglón" en una tabla y así cada método de acceso del objeto (método accesor) es parecido a una "columna" o propiedad de esa tabla que son accesibles gracias a la palabra clave *SELECT*. Cabe recordar que la consulta *SQL* se realiza a través de una cadena, es decir de un *String* la cuál debe ser analizada a través de un objeto del tipo *Query* (incluido dentro de la *API*), proporcionado por el paquete *org.josql* del archivo de referencia mencionado anteriormente (*JoSQL-2.0.jar*), para poder ejemplificar y de esta forma quede más claro sobre cómo puede ser su implementación (ejemplo demostrativo, queda al programador si el uso de esta *API* le es útil o no) se tiene el siguiente código de consola en *Java* que tiene el mismo funcionamiento que el presentado en el apartado anterior (*LINQ*), solo que el criterio de búsqueda es por nombre, el de ordenamiento es a través de su nombre y de forma descendente:

Java

```
import java.util.ArrayList;
import java.util.List;
//paquete de referencia que contiene las funcionalidades de la API de JoSQL
import org.josql.*;

package org.quaere{

//Declaración de la clase
public class usoJoSQL {

public static void main(String[] args) {
        java.io.File archivos= new java.io.File("c:\\JoSQL");
        java.io.File[] objetosFile= archivos.listFiles();

        ArrayList arrayListArchivos=new ArrayList();
        for(File archivo: objetosFile)arrayListArchivos.add(archivo);
       // Obtener una lista de objetos  java.io.File
        List<java.io.File> listaFile = arrayListArchivos;
        String sentencia="SELECT name,length,path FROM java.io.File"+
                         "where name LIKE '%a%' ORDER BY name DESC";
        //Creación de un objeto Query el cuál permitirá analizar, ejecutar
        y obtener los datos de una consulta SQL
       Query consulta = new Query();
       // Analizar la sentencia SQL a usar
       consulta.parse(sentencia);
       //ejecutar la consulta, en donde el parámetro del método execute()
       //se encuentra la fuente de datos sobre la cuál se aplicará la consulta SQL
       //la cuál debe ser del tipo List
       QueryResults resultados= consulta.execute(listaFile);

            List<java.io.File> listaResultado= resultados.getResults();
            System.out.println("Consulta ejecutada= " +sentencia);
```

```
            for (int i=0;i<listaResultado.size();i++){
            // Existe una  Lista por cada petición de método de acceso (get)
            en la sentencia SQL(name,length,path),,
               List r = (List) listaResultado.get (i);
               // El indice 0 contiene el nombre (name)
               System.out.println ("Nombre: " + r.get (0));
               // el indice 1 contiene la longitud (length)
               System.out.println ("Longitud: " + r.get (1));
               // el indice 2 contiene la ruta (path)
               System.out.println ("Path : " + r.get (2));
            }
         System.out.println("------------------------------------");
         System.out.println("Numero de Archivos= " +String.valueOf( listaResultado.size()));
      }
   }
}
```

JoSQL es una forma de usar y expresar consultas *SQL* en el leguaje *Java* para proporcionar un mayor beneficio en la selección de datos, agrupación y operaciones de ordenamiento sobre objetos, por lo que No es una herramienta de acceso a bases de datos, para ello queda al ingenio del equipo desarrollador diseñar una metodología en la cuál se pueda mantener una interacción ininterrumpida y sincronizada entre los datos provenientes de una base de datos (quedando establecida en la capa de acceso a datos) con esta *API* (correspondiente a la capa de la lógica de negocio).

1.3.1 MÉTODOS ACCESORES.

El criterio usado en la *API* de *JoSQL* para acceder a las características de un objeto, se basa en un concepto en particular, un "método accesor". Un método accesor es básicamente una forma para poder conocer los atributos de un objeto. Estos proveen una forma precisa de representar los nombres de los métodos públicos ó propiedades que son accesibles, separados por ".".). Los métodos accesores generalmente siguen la convención en la nomenclatura de JavaBean, por ejemplo. Los métodos de acceso inician con "get" o "is" (en el caso de que el valor a devolver sea booleano) y son seguidos por un nombre dado con la primera letra capitalizada. Sin embargo, existen programadores que no siempre siguen esta convención lo que permite tener nombres de métodos que no inicien con "get" o "is".
Por ejemplo, en el código anterior mostrado se hace uso de la clase *java.io.File*, con lo que se tiene:

- Para acceder al nombre del archivo, se hizo uso de *name*.
- Para acceder a la longitud, se hizo uso de *length*.
- Para acceder a la ruta del archivo: se hizo uso de *path*.

1.3.2 REALIZACIÓN DE LA CONSULTA (QUERY).

La realización de una consulta *SQL* envuelve tres pasos básicos:

1.3.2.1 Analizar (Parse).

En la primera etapa la consulta *SQL* que se encuentra en forma textual es analizada y a partir de esto se genera una representación a través de un objeto en Java y este proceso resulta transparente para el programador, sin embargo si existe un error en la cadena, un error en el análisis es devuelto (no es enviada ninguna excepción):

Java

```
String sentencia="SELECT name,length,path
                  FROM java.io.File
                  WHERE name
                  IKE '%a%'
                  ORDER BY name DESC";

consulta.parse(sentencia);
```

Incorrecto: IKE
Correcto: LIKE

Resultado:

```
--------------ERROR------
Unable to parse query: SELECT name,length,path FROM java.io.File where name
ike '%a%'
ORDER BY name DESC
```

1.3.2.2 Inicialización.

Esta etapa generalmente ocurre inmediatamente después del análisis (puede decirse que prácticamente se encuentran en una misma etapa). En esta etapa la representación en un objeto de Java de la consulta es inicializada y entonces se puede ejecutar la consulta (*Query*) a través del método de instancia *execute* de un objeto *Query*.

1.3.2.3 Ejecución.

A partir de esta tercera etapa, se realiza la ejecución de la consulta. Esta etapa se encuentra de forma separada a las anteriores etapas (análisis/inicialización) y diferente, lo que significa que la consulta puede ser reutilizada. La ejecución de la consulta se realiza a través del método *execute* del objeto mencionado en la etapa anterior (*Query*), este método acepta como parámetro una lista de la cuál se va o quiere realizar la consulta, es decir, que esta parámetro nos servirá como la fuente de datos.

```
consulta.execute(listaFile);
```

1.3.3 CARACTERÍSTICAS SOPORTADAS.

Al momento de escribir esta tesis, *JoSQL* actualmente soporta las siguientes características SQL:

- *SELECT* * (cuyo significado se puede entender como "regresar al propio objeto", puede tomarse como una instancia de la clase, que coincida con la cláusula WHERE o HAVING). Si se toma como punto de referencia los ejemplos presentados anteriormente, en este tema de *JoSQL*, el colocar *SELECT **, se tomaría como obtener a los propios objetos de tipo *java.io.File*.
- *SELECT* columnas (en este caso métodos de acceso *get* ó propiedades), *SELECT* función, *SELECT* expresión (aritmética o booleana), *SELECT* valor a guardar, *SELECT* variable, y otras.

Además de las anteriores, también se encuentran otras características, tal es el caso de las cláusulas que se pueden colocar en la propia sentencia SQL definida en una cadena, las cuales solo se mencionarán y no se explicarán, esto solo para poder ahorrar espacio en el trabajo presente, si se desea buscar mas información sobre estas cláusulas y otras características (expresiones, manejo de variables en la sentencia, etc.), acceder a la página principal localizada en *http://josql.sourceforge.net*.

Cláusulas:

- SELECT
- FROM
- WHERE
- GROUP BY
- LIMIT
- EXECUTE ON
- GROUP BY ORDER
- GROUP BY LIMIT

Expresiones:

- BooleanExpression
- ConstantExpression
- NewObjectExpression
- SubQueryExpression
- AliasedExpression
- ArithmeticExpression
- BetweenExpression
- EqualsExpression
- GTLTExpression
- IsNullExpression
- LikeExpression
- SelectItemExpression
- ExpressionList

1.4 QUAERE.

Quaere es una infraestructura que se distribuye bajo una licencia libre (*Apache 2.0 License*) que incorpora una sintaxis de consulta *SQL* para aplicaciones que estén escritas en Java, lo que permite integrar el lenguaje de consulta *SQL* con código Java. Aunque todavía no hay una versión estable (al momento de escribir esta tesis), y como se muestra en su página, es funcional y capaz de realizar operaciones de consulta sobre fuentes de datos de una manera similar a como se está manejando en la tecnología de Microsoft basada en *LINQ*. Este es un *DSL*[1] interno[2] escrito en Java con lo cuál se pueden realizar un gran número de operaciones de consulta incluyendo restricción, selección, proyección, conjuntos, partición, agrupación, ordenación, cuantificación, agregación y operadores de conversión, sobre un número de colecciones y otras fuentes consultables como arreglos haciendo uso de una sintaxis común y expresiva e implementando la interfaz *Iterable*. Todas estas operaciones se encuentran ubicadas dentro de la *API* de *Quaere*. Para ejemplificar lo anterior, se muestran los siguientes fragmentos de código:

Ejemplo 1.

Java

```
//Seleccionando todos los enteros que sean menores a 5 de un arreglo de enteros
Integer[] numeros={5, 4, 1, 3, 9, 8, 7, 2, 0};
Iterable<Integer> numerosMenores= from("n").in(numeros).
                              where(lt("n",5).
                              select("n");
System.out.println("Todos los números menores que 5 son:")
for (Integer n: numerosMenores)
System.out.println(n);
```

Ejemplo 2:

Java

```
public class usoQuaere {
    public static void main() {
//getCiudades () regresa un arreglo de objetos Ciudad
        Ciudad[] ciudades=Ciudad.getCiudades();
        Iterable<Ciudad> poblaciones =from("ciudad").in(ciudades).
                                orderBy("Ciudad.getPoblacion()").
                                 select("ciudad");
        for (Ciudad ciudad : poblaciones)
            System.out.println(city);
    }
}
```

Nota: Como se puede observar, la implementación de la interfaz Iterable le permite a un objeto ser iterado con el uso de la sentencia "for (Tipo nombre: colección_de_objetos)"

Quaere permite trabajar también sobre los datos extraídos desde una base de datos, para ello se tendría que realizar una metodología en la que se incluya como primer paso la obtención de los datos relaciones de la base de datos para posteriormente realizar operaciones de consulta haciendo uso de esta *API*. Para poder desarrollar y ejecutar consultas con esta *API*, se debe contar con la versión del archivo *Quaere*, la cuál no se encuentra en su versión final y estable por lo que aún no ha sido liberado demás de que se menciona en su página que es susceptible a cambios. *Quaere* depende de la contribución de los programadores que estén interesados en su desarrollo y mejora, por lo tanto, para empezar a formar parte en el desarrollo de *Quaere* se puede obtener el código fuente a través de una subversión contenida en el repositorio *http://svn.codehaus.org/quaere* (en esta dirección se tiene acceso anónimo).

[1] Domain Specific Language, Lenguaje Específico de Dominio a veces confundido con Lenguaje de Dominio Específico. Es un lenguaje de programación que ataca a un problema en particular.

[2] esto es porque en vez de definir un nuevo lenguaje, se moldea el lenguaje de propósito general, en esta caso Java, dentro del DSL. También es conocido como DSL encajado

1.5 ¿POR QUÉ USAR LINQ?

Se acaba de tener una visión general de *LINQ*. Las grandes preguntas en este punto son:

- ¿Qué diferencias existen entre *LINQ* y *JoSQL*<-> *Quaere*?
- ¿Por qué requerir una herramienta como *LINQ*?
- ¿*LINQ* fue creado sólo para trabajar con los lenguajes de programación, datos relacionales y *XML*, y con esto se trata de hacerlo más útil?

Para ello se partirá de un hecho simple que incurre de forma recurrente en el desarrollo de las aplicaciones de software y que ha sido considerado en el origen de *LINQ*: La gran mayoría de las aplicaciones que se desarrollan e implementan acceden a datos o interaccionan con una base de datos relacional. En consecuencia, para programar aplicaciones que se centran en este elemento primordial, el aprendizaje de un lenguaje como *C#* no es suficiente. Además de saber el propio lenguaje en el que se desarrollará gran parte del sistema, también se tiene que aprender otro lenguaje como *SQL* esto debido a que el programador tiene que cambiarse de escribir código en *C#* a escribir comandos de *SQL*, en cuyo caso se encuentran basados en comandos *String*. No solamente en este caso se necesita saber varios lenguajes de programación sino también buenos conocimientos de aplicar las *API's* que accedan a los distintos *SMBD* (sistemas manejadores de base de datos) que se conectan en dominios diferentes, para crear totalmente la aplicación. Se empezará por echar un vistazo a algunos códigos mostrados anteriormente y compararlo con su similar en *LINQ*, posteriormente se mostrará un fragmento de código que accede a una base datos y que utiliza la *API* estándar para *.NET Framework* (*ADO .NET*), la cuál es similar a su contrapartida en *Java* (*JDBC*). Esto nos permitirá señalar las diferencias y dificultades comunes que se encuentran en este tipo de código. Se observará que *LINQ* manipula la incongruencia existente entre las fuentes de datos y el propio lenguaje de programación. Por último, un breve ejemplo de código dará una idea de cómo *LINQ* es la solución al problema.

1.5.1 LINQ FRENTE A JOSQL Y QUAERE.

Primero hay que mencionar que las expresiones de consulta con *LINQ* en los lenguajes de *C# 3.0* y *VB .NET* son vistos por el programador como una sintaxis con declaraciones de palabras reservadas tales como *from*, *in*, *where*, *select*, etc., aunque en realidad son llamadas a métodos extensores que son realizadas por el compilador del propio lenguaje (y que son predefinidas por los proveedores), está el hecho de que, al estar soportado directamente en los lenguajes, *LINQ* ofrece una mayor profundización en la construcción de la lógica del negocio, de lo que puede lograrse mediante el uso exclusivo de librerías, como es el caso de los proyectos ó trabajos que se encuentran disponibles en Java como *JoSQL* y *Quaere*. En el caso de *JoSQL*, la sentencia *SQL* que debe crearse es del tipo cadena (*String*) y esta debe ser analizada a través del método de instancia de *Query*, por lo que aquí se encuentra la primera desventaja, ¿qué sucede si no es correcta la cadena?, ¿qué pasa si el método al que se quiere acceder no existe?, como respuesta a eso se da un error:

Java

```
String sentencia="SELECT name,length,path"+
                 "FROM java.io.File"+
                 "where name like '%a%' ORDER BY name DESC";
Query consulta = new Query();
onsulta.parse(sentencia);
QueryResults resultados= consulta.execute(listaFile);
```

1. Consulta SQL en una cadena (string)
¿Existe el método de acceso name,lenght ó path?
¿Esta bien escrito?

El error solo se descubre al ejecutar la aplicación a través del método parse()

Los métodos accesores o propiedades deben estar presentes en la clase del objeto devuelto.

Como se puede observar, no existe la seguridad de que la cadena sea la adecuada, a falta de una comprobación en tiempo de compilación. La diferencia fundamental es que en *LINQ* las consultas que se desarrollan son fuertemente tipadas (existe una seguridad de tipos, dando como resultado que las propiedades del objeto estén disponibles en la consulta), las operaciones dentro del *where*, *orderby* y *select* son expresiones reales y tienen inferencia de tipos, a diferencia de *JoSQL* que en primera instancia debe realizar un análisis del texto contenida en una cadena que representa a la consulta y por lo tanto no puede descubrirse errores hasta tiempo de ejecución. A continuación se muestra un fragmento de código tomado del apartado 1.1.1 (*¿Qué es LINQ?)* para ejemplificar lo anterior:

C#

```
FileInfo[] objetosFile = new DirectoryInfo(@"C:\LINQ").GetFiles();

var seleccionarFiltrado = from archivo in objetosFile
                          where archivo.Name.contains("a")
                          orderby archivo.Length descending


foreach (var res in seleccionarFiltrado)
   {
     Console.WriteLine("Nombre: " + res.Extension);
     Console.WriteLine("Longitud: " + res.Length);
     Console.WriteLine("Path: " + res.FullName);
     Console.WriteLine("\n");
   }
```

Soporte para declarar palabras reservadas reales (from, in , where, orderby, select)

Al ser la variable archivo un objeto del tipo FileInfo, están disponibles todos sus miembros públicos

Inferencia de tipo

No ocurre un error al no poder escribir métodos o propiedades que no existan en el objeto devuelto

Además para poder manejar datos extraídos de una base de datos, es indispensable hacer uso explícitamente de la *API* y los controladores que permita la ejecución de operaciones sobre bases de datos desde el lenguaje de programación (*JDBC*), pero con *LINQ* esto ya se tiene de forma integrada a *ADO. NET*, para esto se tendría que realizar primero el proceso de conexión a BD, extracción o selección de los datos requeridos y su manipulación a través de la *API* de *JoSQL*. Un entorno ideal para la creación de aplicaciones de software consistiría en permitir a los programadores describir la lógica de negocio y el estado del dominio del problema que están modelando con un mínimo o ningún "ruido" proveniente de la representación subyacente y de la infraestructura que la soporta. Las aplicaciones deben ser capaces de interactuar con los conjuntos de resultados para mantener un estado persistente de los datos del sistema en los términos del problema. Cabe mencionar que en el caso de *Quaere*, la metodología que usa es casi similar a la de *LINQ*, se hace implementación de la interfaz *Iterable<T>* para poder realizar un ciclo sobre los datos resultantes, al igual que en *LINQ* se hace uso de la Interfaz genérica *IEnumerable<T>:*

Java

```
Integer[] numeros={5, 4, 1, 3, 9, 8, 7, 2, 0};
Iterable<Integer> numerosMenores= from("n").in(numeros).
                                  where(lt("n",5).
                                  select("n");

System.out.println("Todos los números menores que 5 son:")
for (Integer n: numerosMenores) {
System.out.println(n);
}
```

Métodos:

- from()
- in()
- where()
- select()

Pero otra diferencia fundamental es que se utilizan métodos, los cuáles están muy bien definidos pero a diferencia de *LINQ*, este último extiende a *.NET Framework* para dar el soporte en la declaración de nuevas palabras claves como *from*, *where*, *select*, etc.

LINQ además es una mejora al lenguaje, no solo una adición de una nueva API/librería, para ello se ha tenido que modificar el lenguaje y el compilador. *Quaere* sigue usando literales en cadenas para el criterio en la consulta *SQL* y en la proyección. Mientras que *LINQ* en *C#* usa expresiones lambda. Por estas razones, es que se ha elegido como tema principal de esta tesis *LINQ* y en específico *LINQ To SQL*. De ahora en adelante se abarcaran los conceptos que engloba esta tecnología.

1.5.2 RESOLUCIÓN DE PROBLEMAS CON LINQ.

Actualmente en las aplicaciones de software que atienden la demandas de las empresas se tiene un frecuente uso de bases de datos en las que *.NET Framework* y otras plataformas se aborda la necesidad de hacer uso de *API´s* que puedan tener acceso a los datos almacenados. Por supuesto, este ha sido el caso desde que se dio la primera aparición de *.NET*. La librería de clases de *.NET Framework* incluye *ADO.NET*, que proporciona una API para acceder a bases de datos relacionales y para representar datos relacionales en memoria. Esta API se compone de clases como *SqlConnection*, *SqlCommand*, *SqlReader*, *DataSet* y *DataTable*, por nombrar algunos. El problema con estas clases es que se fuerza al programador a trabajar explícitamente con tablas, registros y columnas, mientras que los lenguajes modernos, tales como *C #, VB.NET* o incluso *Java* usan paradigmas orientado a objetos. Ahora que el paradigma orientado a objetos es el modelo predominante en el desarrollo de software, los desarrolladores incurren en un complejas técnicas para poderlos abstraer, específicamente las bases de datos y *XML*.

El resultado es que gran parte del tiempo se dedica a la escritura de código que no solo se centra en la capa de negocio sino también en lo referente a la capa de acceso a datos, removiendo esto, podría incrementarse la productividad en la programación intensiva sobre los datos, pero no solo en eso, sino también en la calidad. En el siguiente ejemplo se muestra un simple código de cómo se suele acceder a una base de datos en un programa orientado a objetos, en este caso *C#*:

C#

```
//connectionString es una cadena de conexión válida para una base de datos  de SQL Server
 using (SqlConnection connection = new SqlConnection("connectionString"))
  {
      connection.Open();
//creación de un SqlCommand, el cuál nos servirá para poder representar
//sentencias SQL ó procedimientos almacenado y ser ejecutadas
//en el servidor
      SqlCommand command = connection.CreateCommand();
      command.CommandText =
      @"SELECT Nombre, Pais FROM Clientes WHERE Ciudad = @Ciudad";
      //se le agrega un parámetro de cadena al SqlCommand
      command.Parameters.AddWithValue("@Ciudad", "Paris");
      //Se envía la consulta SQl ala base de datos y se regresa un
      //SqlDataReader, el cuál ofrece una manera de leer los datos (o renglones)
      //de una tabla solo hacia adelante
      using (SqlDataReader reader = command.ExecuteReader())
      {
          //Mientras haya datos en el SqlDataReader, se realizará la lectura
          //de los datos y serán asignados a una variable, el método Read()
          //realiza en avance del SqlDataReader hacia el siguiente elemento
          while (reader.Read())
          {
//se obtienen los datos del SqlDataReader, en donde
//el método GetString() devuelve el valor de la columna especificada como
//parámetro del tipo int, o como daena si se sabe el nombre de la columna
              string nombre = reader.GetString(0);
              string pais = reader.GetString(1);
          }
      }
  }
```

1. Consulta SQL en una cadena (string)

2. Parámetros vagamente vinculados

3. Columnas vagamente tipadas (typed)

- A pesar de que desea llevar a cabo una tarea sencilla, son obligatorios varios pasos y diferentes detalles en el código.
- Las consultas se expresan como cadenas entre comillas (1), lo que significa que todos los tipos son evaluados después de mandar la consulta al servidor de la base de datos. ¿Qué ocurre si la cadena no es válida?, cuya respuesta es bien conocida por todos los programadores.
- Lo mismo se aplica para los parámetros (2) y los *ResultSet* (3): son vagamente definidos. ¿Son las columnas del tipo que esperamos? Además, ¿estamos seguros de estamos usando el número correcto de parámetros? ¿Están los nombres de los parámetros en concordancia entre la consulta y las declaraciones del parámetro?.

La idea principal es que mediante el uso de *LINQ* el programador sea capaz de obtener acceso a cualquier fuente de datos a través de la realización de consultas como la mostrada en el código anterior, directamente en el lenguaje de programación:

```
from variable in Clientes
where Clientes.Nombre.StartsWith("A") && Clientes.Orden.Count > 0
orderby Clientes.nombre
select new { Clientes.Nombre, Clientes.Orden }
```

En esta consulta, los datos se encuentran en memoria ya se que provienen del resultado de una operación en el lenguaje, de una base de datos, de un documento *XML*, o de otro lugar, en este caso la sintaxis sigue siendo similar en cuanto a la manipulación de los objetos que provienen de diversas fuentes, pero no exactamente la misma, esto debido a las definiciones (clases y métodos) que existen para cada una de estas extensiones de *LINQ*. Este tipo de consulta se puede utilizar con múltiples tipos de datos y diferentes fuentes de datos, gracias a las características de extensibilidad de *LINQ*.

1.6 FUNDAMENTOS DE LA INFRAESTRUCTURA . NET FRAMEWORK.

1.6.1 CARACTERÍSTICAS PRINCIPALES.

.NET Framework es la infraestructura adyacente de una nueva plataforma que se centra en la transparencia de redes aunado a una independencia de plataforma de hardware para que se proporcione una rápida, segura y sólida manera de crear y desarrollar aplicaciones, con lo que se permita una integración más rápida, ágil y un acceso más simple a todo tipo de información.

Con esto se genera un entorno integral de desarrollo y ejecución multilenguaje de componentes que proporciona los bloques básicos para desarrollar y ejecutar Aplicaciones Web y Servicios Web. El diseño de *.NET Framework* está enfocado en los siguientes objetivos:

- Provee un ambiente encaminado a la programación orientada a objetos, en el que el código de los objetos se pueda almacenar y ejecutar de forma local pero distribuida en Internet o ejecutar de forma remota.
- Proporciona un ambiente que une versiones anteriores para la ejecución de código, y así eliminar los conflictos y la reducción en la implementación de cualquier software. Además de que este entorno fomente la ejecución para el código de terceros y elimine los problemas de rendimiento ocasionados por hacer uso de intérpretes de comando.
- Ofrece al programador una herramienta en la que se pueda convivir con tipos de aplicaciones muy diferentes, como las basadas en *Windows* (*Stand-Alone*) o en la que están basadas en la Web (*ASP .NET*).
- Basar toda la comunicación en estándares usados para asegurar y permitir que el código de *.NET Framework* se pueda integrar con otros tipos de código de plataformas orientadas a objetos.

En las secciones siguientes, se revisarán las capas correspondientes de *.NET Framework*.

1.6.2 CLR (COMMON LANGUAGE RUNTIME)

El *Common Language Runtime* (*CLR*) es el núcleo de la plataforma *.NET*. Administra el código en tiempo de ejecución y proporciona los servicios centrales para la plataforma, como la administración de memoria, la administración de subprocesos y la interacción remota, al tiempo que aplica una gran seguridad a los tipos y otras formas de especificación del código. Las principales características y servicios que ofrece el *CLR* son:

1.6.2.1 Ejecución multiplataforma.

El *CLR* actúa como una máquina virtual, encargándose de ejecutar las aplicaciones diseñadas para la plataforma en *.NET*. Es decir, cualquier plataforma para la que exista una versión del *CLR* podrá ejecutar cualquier aplicación de *.NET*. *Microsoft* ha desarrollado versiones del *CLR* para la mayoría de las versiones de *Windows*: *Windows 95, Windows 98, Windows ME, Windows NT 4.0, Windows 2000, Windows XP* y *Windows Vista* (que puede ser usado en CPUs que no sean de la familia x86). Por otro lado también hay terceros que están desarrollando de manera independiente versiones de libre distribución del *CLR* para *Linux*.

1.6.2.2 Integración de lenguajes.

Para todos los lenguajes de *.NET Framework* existe un compilador que genera código para la misma plataforma, con lo que es posible que cualquier código generado por cualquier lenguaje de *.NET Framework* se ejecutado en la plataforma. *Microsoft* ha desarrollado un compilador de *C#* que genera código de este tipo, así como versiones de sus compiladores de *Visual Basic* (*Visual Basic.NET*) y *C++* (*C++* con extensiones gestionadas) que también lo generan y una versión del intérprete de *JScript* (*JScript.NET*) que puede interpretarlo. La integración de lenguajes es tal que es posible escribir una clase en *C#* que herede de otra escrita en *Visual Basic.NET* que, a su vez, herede de otra escrita en *C++* con extensiones gestionadas.

1.6.2.3 Gestión de memoria.

La asignación de memoria a instancias de objetos *.NET* se realiza de forma contigua desde el "*heap*" de la memoria administrada (objetos administrados, una asociación de memoria administrada por el *CLR*). Cuando no exista una referencia a un objeto, y ésta no pueda ser extendida ó usada, esta es enviada a la basura. Sin embargo, esta todavía permanece en la memoria a la cuál fue asignada. El *CLR* incluye un recolector de basura (*Garbage Collector, GC* por sus siglas en inglés) que evita que el programador tenga que tener en cuenta cuándo ha de destruir los objetos que dejen de serle útiles. El *GC* usado por el *.NET Framework* actualmente es generacional. Los objetos son asignados a una generación; aquellos que son creados recientemente pertenecen a la Generación 0. Los objetos que sobreviven a la recolección de basura son etiquetados como Generación 1, y los objetos de la Generación 1 que sobreviven a otra recolección de basura pasan a pertenecer a la Generación 2. .N*ET Framework* hace uso de los objetos de la Generación 2, esto significa que los objetos que pertenecen a una Generación más alta, son menos frecuentes a sufrir el efecto de una recolección de basura que los objetos que se encuentran en generaciones inferiores.

1.6.2.4 Seguridad de tipos.

El *CLR* realiza una comprobación en las conversiones de tipos que se van realizando durante la ejecución del programa, de esta forma se va facilitando la detección de errores de programación que comúnmente se generan y que resultan difíciles de encontrar.

1.6.2.5 Aislamiento de procesos.

El *CLR* proporciona un nivel de seguridad alto entre los procesos, con esto se evita que cierto código proveniente de un proceso pueda tener acceso al código de otro proceso, sin esto se impide que unos procesos puedan atacar a otros. Esto se logra con la seguridad de tipos, ya que se evita que se pueda convertir un objeto a cierto tipo de mayor tamaño que el propio, ya que

podría tenerse acceso a espacios en memoria que no le pertenecen pero que podrían pertenecer a otro proceso, además de que no se permite acceder a posiciones arbitrarias de memoria.

1.6.2.6 Manejo de excepciones.

En el *CLR* maneja los errores que se puedan producir durante la ejecución de una aplicación a través de excepciones. Esta forma de trabajar es diferente a como se hacia en los sistemas de Windows antes de la aparición de la plataforma *.NET*, donde algunos errores se manejaban a través de códigos de error en formato *Win32*, mediante *HRESULTs*[3] y otros mediante excepciones. El *CLR* permite que excepciones lanzadas desde código para *.NET* escrito en un cierto lenguaje se puedan capturar en código escrito usando otro lenguaje, e incluye mecanismos de depuración que pueden saltar desde código escrito para *.NET* en un determinado lenguaje a código escrito en cualquier otro.

1.6.2.7 Soporte multihilo.

El *CLR* es capaz de trabajar con aplicaciones divididas en múltiples hilos que pueden separarse a lo largo de la ejecución de una aplicación, según el número de procesadores de la máquina sobre la que se ejecuten. Las aplicaciones pueden crear nuevos hilos, destruirlos, suspenderlos por un tiempo o hasta que les llegue una notificación, enviarles notificaciones, sincronizarlos, etc.

1.6.2.8 Seguridad avanzada.

El *CLR* limita la ejecución de códigos de acuerdo a su origen o del usuario que los ejecutó. Con esto, el nivel de libertad con el que puede trabajar un código que proviene de Internet es diferente al código que se encuentra de forma local o al procedente de una red local, con esto puede no darse los mismos permisos a un mismo código según el usuario que lo esté ejecutando o el rol que tiene. Con esto se asegura la computadora del cliente, para que el código que se esté ejecutando no pueda poner en peligro sus archivos.

1.6.2.9 Interoperabilidad con código antiguo.

Se integran mecanismos indispensables que proporcionan una compatibilidad entre código escrito para la plataforma *.NET* y código escrito previo a la aparición de la misma y, por tanto, no preparado para ser ejecutando dentro de ella. Con esto se proporciona el acceso a objetos *COM*[4] como el acceso a funciones en *DLL's* preexistentes (como la *API Win32*). El *CLR* también es compatible con el software actual y el software antiguo. La interoperabilidad entre el código administrado y no administrado permite que los desarrolladores continúen utilizando los componentes *COM* y las *DLL* que necesiten.

1.6.3 COMMON TYPE SYSTEM (CTS).

El *Common Type System* (CTS) o Sistema de Tipos Común es el conjunto de reglas que deben de seguirse al momento de la definición de los diversos tipos de datos para el *CLR*. Es decir, aunque cada lenguaje de *.NET* disponga de su propia sintaxis para definir los diferentes tipos de datos existentes, en el *CIL* resultante de la compilación de sus códigos fuentes, se deben de cumplir las reglas que se imponen en el *CTS*. Algunos ejemplos de estas reglas son:

[3] HRESULT es el tipo de datos que sirve para indicar el estado de las operaciones en los componentes de Microsoft. Tiene un valor de 32 bits divididos en tres campos: un código de gravedad, un código de instalación, y un código de error. El código de gravedad indica si el valor de retorno representa la información, advertencia o error. El código de instalación identifica el área del sistema responsable del error. El código de error es un número único que es asignado para representar a la excepción. Cada excepción se asigna a un HRESULT.

[4] Component Object Model (COM): es una plataforma de Microsoft para componentes de software introducida por dicha empresa en 1993 y es utilizada para permitir la comunicación entre procesos y la creación dinámica de objetos, en cualquier lenguaje de programación que soporte dicha tecnología.

- Cada tipo de dato puede constar de cero o más miembros. Cada uno de estos miembros puede ser un campo, un método, una propiedad o un evento.
- No puede haber herencia múltiple, y todo tipo de dato ha de heredar directa o indirectamente de *System.Object*.
- Los modificadores de acceso admitidos son:

Modificador	Código desde el que es accesible el miembro
public	Cualquier código
private	Código del mismo tipo de dato
family	Código del mismo tipo de dato o de hijos de éste.
assembly	Código del mismo ensamblado
family and assembly	Código del mismo tipo o de hijos de éste ubicado en el mismo ensamblado
family or assembly	Código del mismo tipo o de hijos de éste, o código ubicado en el mismo ensamblado

Modificadores de acceso a miembros admitidos por el CTS

1.6.3.1 Clasificación de tipos.

El sistema de tipos común se divide en dos categorías generales de tipos, los cuáles a su vez están divididos en subcategorías:

1.6.3.1.1 Tipos de valor

Estos tipos contienen directamente los datos y las instancias de los tipos que se definen se asignan en la pila o se asignan en línea en una estructura. Los tipos de valor pueden ser integrados (implementados por el motor en tiempo de ejecución), definidos por el usuario o enumeraciones. Estos tipos contienen una copia de los datos y con esto las variables de las demás copias no se ven afectadas en las operaciones.

1.6.3.1.2 Tipos de referencia

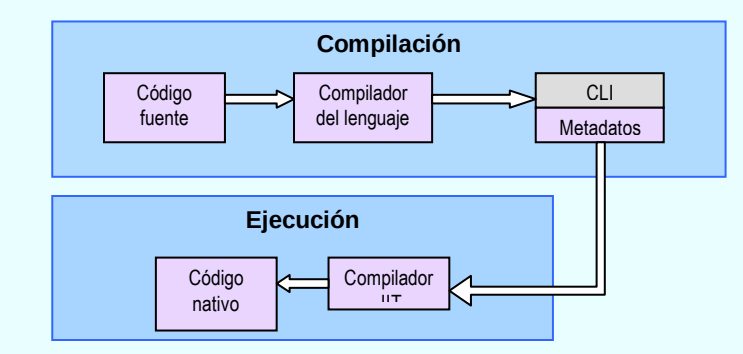
Estos almacenan una referencia que apunta hacia la dirección de memoria del valor y se asignan al montón (espacio de memoria). Estos tipos de referencia pueden ser tipos auto descriptivos, de puntero o de interfaz. Este tipo se puede determinar a partir de los valores que contienen los tipos auto-descriptivos y estos a su vez, se dividen en matrices y tipos de clase. Los tipos de clase son clases definidas por el usuario, tipos de valor a los que se ha aplicado la conversión *boxing* y delegados.

Las variables que son tipos de referencia pueden hacer referencia al mismo objeto y, por lo tanto, las operaciones en una variable pueden afectar al mismo objeto al que hace referencia otra variable.

1.6.4 COMMON INTERMEDIATE LANGUAGE (CIL).

El *Common Intermediate Language* es un lenguaje de programación de bajo nivel entendible y fácil de comprender por los humanos. El propósito del *CIL* es proporcionar una plataforma para un lenguaje neutro hacia el desarrollo y la ejecución de aplicaciones de software, en las que se incluyen funciones para la manipulación de excepciones, recolección de basura, seguridad e interoperabilidad. El resultado del proceso de compilación del código fuente proveniente de cualquiera de los lenguajes soportados por *.NET*, a un código intermedio es el *CIL*. Para generarlo, el compilador se basa en la especificación *CLS* (*Common Language Specification*) que establece las reglas necesarias para generar el código *CIL* que resulte compatible con el *CLR*.

La ventaja de este lenguaje intermedio es que proporciona una ejecución multiplataforma y una integración entre los lenguajes al ser independiente de la computadora, además de un formato común para el código máquina generado por todos los compiladores que generen código para *.NET*. Sin embargo, dado que las computadoras no pueden ejecutar directamente *CIL*, antes de ejecutarlo tiene que convertirse a código nativo de la computadora sobre la que se vaya a ejecutar. Para poder realizar esta tarea existe un elemento del *CLR* conocido como compilador *JIT* (*Just-In-Time* ó en tiempo de ejecución) que va convirtiendo dinámicamente el código *CIL* a ejecutar en código nativo según sea necesario. Éste es el que genera el código máquina real que se ejecuta en la plataforma del cliente. De esta forma se consigue una independencia con la plataforma de hardware. La compilación *JIT* la realiza el *CLR* a medida que el programa invoca métodos. El código ejecutable obtenido se almacena en la memoria caché de la computadora, siendo recompilado de nuevo sólo en el caso de producirse algún cambio en el código fuente[5].



Compilación de código fuente a código nativo

1.6.5 COMMON LANGUAGE SPECIFICATION (CLS).

El *Common Language Specification* (*CLS*) o Especificación del Lenguaje Común es una serie de reglas que deben seguirse en el momento de definir tipos a través de un lenguaje de *.NET* si se desea que sean accesibles desde cualquier otro lenguaje gestionado. Por otra parte, si no importa la interoperabilidad entre lenguajes tampoco es necesario seguirlas. A continuación se listan algunas reglas importantes del *CLS*:

- Los tipos de datos básicos admitidos son *bool*, *char*, *byte*, *short*, *int*, *long*, *float*, *double*, *string* y *object*. Además, no es necesario que todos los lenguajes deban admitir los tipos básicos como los enteros sin signo o el tipo decimal como lo hace *C#*.
- Las tablas deben de tener una o más dimensiones, y el número de dimensiones de cada tabla ha de ser fijo. Además, el primer índice con el que se inicia es desde 0.
- Se pueden definir tipos abstractos y tipos sellados. Los tipos sellados no pueden tener miembros abstractos.
- Las excepciones derivan de *System.Exception*, los delegados de *System.Delegate*, las enumeraciones de *System.Enum*, y los tipos por valor que no sean enumeraciones de *System.ValueType*.
- En un mismo código no se pueden definir varios identificadores cuya única diferencia estribe en la capitalización usada en sus nombres. Con esto se evitan problemas al acceder a ellos usando lenguajes no sensibles a mayúsculas (*case sensitive*).
- Las enumeraciones no pueden implementar interfaces, y todos sus campos deben ser estáticos y del mismo tipo. El tipo de los campos de una enumeración sólo puede ser uno de los siguientes tipos básicos: *byte*, *short*, *int* o *long*.

[5] Las llamadas subsecuentes al código ejecutará solamente la versión en la memoria caché. Una vez que la aplicación sea terminada y se inicia de nuevo, este proceso se repite.

1.6.6 BASE CLASS LIBRARY (BCL).

La biblioteca de clases base (*BCL* de ahora en adelante), es otro de los elementos principales que forman parte en la estructura de *.NET Framework*, la cuál es una colección de clases (lo que proporciona tipos con los que a través de su propio código administrado se pueden derivar funciones) que se encuentran íntimamente relacionadas con el *CLR*, para que puedan ser reutilizados y empleados en diversos escenarios para el desarrollo de software. La *BCL* se encuentra organizada a través de una jerarquía de espacios de nombres (*namespace*) que agrupan clases con funciones similares. Esta librería de *.NET* está disponible para todos los lenguajes en *.NET*. Por ejemplo, puede utilizar *.NET Framework* para desarrollar los siguientes tipos de aplicaciones y servicios:

- Aplicaciones de consola
- Aplicaciones *GUI* de *Windows* (formularios *Windows Forms*)
- Aplicaciones de *ASP.NET*
- Servicios Web *XML*

Por ejemplo, los espacios de nombres más usados son:

Espacio de nombres	Utilidad de los tipos de datos que contiene
System	Tipos muy frecuentemente usados, como los tipos básicos, tablas, excepciones, fechas, números aleatorios, recolector de basura, entrada/salida en consola, etc.
System.Collections	Colecciones de datos de uso común como pilas, colas, listas, diccionarios, etc.
System.Data	Manipulación de bases de datos. Forman la denominada arquitectura ADO.NET.
System.IO	Manipulación de ficheros y otros flujos de datos.
System.Net	Realización de comunicaciones en red.
System.Reflection	Acceso a los metadatos que acompañan a los módulos de código (reflección).
System.Runtime.Remoting	Acceso a objetos remotos.
System.Security	Acceso a la política de seguridad en que se basa el CLR.
System.Threading	Manipulación de hilos.
System.Web.UI.WebControls	Creación de interfaces de usuario basadas en ventanas para aplicaciones Web.
System.Windows.Forms	Creación de interfaces de usuario basadas en ventanas para aplicaciones estándar.
System.XML	Acceso a datos en formato XML.

Espacios de nombres de la BCL más usados

Por ejemplo, las clases de formularios *Windows Forms* son un conjunto de clases reutilizables que simplifican de gran manera el desarrollo de interfaces *GUI* (*Graphical User Interface*, Interfaz gráfica de usuario) para *Windows*. Si se desarrolla una aplicación *Web Form* de *ASP.NET*, pueden hacerse uso de las clases de formularios *Web Forms*.

La *BCL* también contiene un pequeño subconjunto de la Biblioteca de Clases y es el núcleo de las clases que sirven como la *API* básica del *CLR*. Las clases localizadas en los ensamblados mscorlib.dll y algunas de las que se encuentran en *System.dll* y *System.core.dll* son consideradas partes de la *BCL*. También existe un conjunto expandido de librerías, en las que se encuentran *WinForms, ADO.NET, ASP.NET, Language Integrated Query (LINQ), Windows Presentation Foundation, Windows Communication Foundation* entre otros. Las clases de formularios W*indows Forms* contenidas en *.NET Framework* están diseñadas para utilizarse en el desarrollo de *GUI*.

A*SP.NET* es el entorno host que permite a los programadores utilizar *.NET Framework* dándole una orientación hacia aplicaciones para la Web, además *ASP.NET* es una completa arquitectura para el desarrollo de sitios Web y objetos distribuidos en Internet mediante código administrado.

1.6.7 VERSIONES

Microsoft inició el desarrollo de *.NET Framework* a finales de los 1990's, al principio de se encontraba denominada bajo el nombre de *Next Generation Windows Services* (*NGWS*). Pero posteriormente, a finales de 2000, la primera de las versiones beta de. *NET* 1.0 fueron publicadas.

Versión	Número de versión	Fecha de publicación
1	1.0.3705.0	13/02/2002
1.1	1.1.4322.573	24/04/2003
2	2.0.50727.42	07/11/2005
3	3.0.4506.30	06/11/2006
3.5	3.5.21022.8	19/11/2007

Versiones de .NET Framework

1.6.8 .NET FRAMEWORK 1.0 Y .NET FRAMEWORK 1.1

*.NET Framework 1.*0 fue la primera versión que fue lanzada para *.NET Framework* a principios del 2002 y se encuentra disponible para *Windows 98, NT 4.0, 2000, y XP*. *Microsoft* continuó dando soporte para esta versión hasta mediados del año 2007, pero este soporte fue extendido hasta mediados de 2009.

Posteriormente el 3 de abril de 2003 se dio la primera actualización para esta versión la cuál se incluyó como parte del sistema operativo *Windows* y *Windows Server 2003* de *Microsoft*, cuya versión fue la 1.1 (*.NET Framework 1)*. Está nueva versión de *.NET Framework 1.0* se puede encontrar disponible tanto como un paquete redistribuible como un kit de desarrollo de software. Esta versión (1.1) también forma parte de la segunda publicación de *Microsoft Visual Studio .NET*. El soporte para la versión de *.NET Framework 1.1* caducó el 14 de Octubre de 2008, pero se continuará dando soporte hasta el 8 de Octubre de 2013 por parte de *Microsoft* y para *Windows Server 2003* será el 30 de junio del mismo año.

En la versión 1.1 suceden modificaciones tales como:

- Crear y generar de forma interna controles *ASP.NET*.
- Se permite que ensamblados creados con *Windows Forms* se puedan ejecutar desde Internet de forma más confiable, además de permitir el acceso a código de seguridad en aplicaciones *ASP.NET*.
- Manejo de bases de datos *ODBC* y *Oracle* de forma interna ya que anteriormente se encontraba como un complemento para *.NET Framework 1.0*.
- *.NET Compact Framework*, una versión de *.NET Framework* para dispositivos pequeños.
- Soporte para el protocolo de Internet versión 6 (*IPv6*).
- Numerosos cambios en la *API* de *.NET Framework*.

1.6.9 .NET FRAMEWORK 2.0

Fue publicado junto con *Visual Studio 2005, Microsoft SQL Server 2005*, y *BizTalk 2006*. El paquete redistribuible 2.0 (publicado el 22 de Enero de 2006) puede ser descargado de forma gratuita desde la página oficial *Microsoft*.

El kit de desarrollo de software 2.0 (*Software Development Kit SDK*) también se encuentra de forma gratuita desde la página oficial de Microsoft. Ésta está incluida como parte del *Visual Studio 2005* y *Microsoft SQL Server 2005*. Esta versión es la última con soporte para *Windows 2000, Windows 98* y *Windows Me*. A continuación se proporciona información sobre algunas de las principales adiciones y modificaciones que son de de suma importancia para poder dar soporte a *LINQ*:

1.6.9.1 Genéricos.

Estos tipos incorporan el concepto de parámetros de tipo (*T*) a *.NET Framework*, lo cual permite diseñar clases y métodos que aplazan la especificación de uno o más tipos hasta que el código de cliente declara y crea una instancia de la clase o del método. Los tipos genéricos proporcionan reutilización, seguridad de tipos y eficacia de una manera eficaz que sus homólogos no genéricos no pueden. Los tipos genéricos se utilizan comúnmente con las colecciones y los métodos que funcionan en ellas. La versión 2.0 de la *BCL* de *.NET Framework* proporciona un nuevo espacio de nombres, *System.Collections.Generic*, que contiene varias clases nuevas de colección basadas en tipos genéricos. Por ejemplo, mediante la utilización de un parámetro de tipo genérico *T*, se puede escribir una clase única que otro código de cliente puede utilizar sin generar el costo o el riesgo de conversiones en tiempo de ejecución u operaciones de conversión *boxing*, como se muestra a continuación:

```
// Declarar la clase generica
public class ListaGenerica<T>:Ilist<T> {
    void Add(T input) { }
}
class PruebaListaGenerica{
    //declaraciónde una clase interna (anidada)
    private class ClaseEjemplo { }
    static void Main(){
        // Declaración de una lista del tipo int
        ListaGenerica<int> lista1 = new ListaGenerica<int>();
        // Declaración de una lista del tipo string
        ListaGenerica<string> lista2 = new ListaGenerica<string>();
        // Declaración de una lista del tipo ClaseEjemplo
        ListaGenerica<ClaseEjemplo> lista3 =
                                    new ListaGenerica<ClaseEjemplo>();
    }
}
```

Los tipos genéricos proporcionan la solución a una limitación de las versiones anteriores de *CLR* y del lenguaje *C#*, en los que se realiza una conversión de tipos a y desde el tipo base universal *Object*. Con la creación de una clase genérica, se puede crear una colección que garantiza la seguridad de tipos en tiempo de compilación. Las limitaciones del uso de clases de colección no genéricas se pueden demostrar escribiendo un breve programa que utiliza la clase de colección *ArrayList* de la biblioteca de clases base (*System.Collections*) de *.NET Framework*. *ArrayList* es una clase de colección muy conveniente, que se puede utilizar sin modificar para almacenar referencias o tipos de valor.

```
// La forma en .NET Framework 1.1 para crear una lista a través de un objeto Arraylist:
System.Collections.ArrayList lista1 = new System.Collections.ArrayList();
lista1.Add(3);
lista1.Add(105);

System.Collections.ArrayList lista2 = new System.Collections.ArrayList();
lista2.Add("CADENA 1");
lista2.Add("CADENA 2");
```

Sin embargo existe un problema que resulta invisible a simple vista. Cualquier referencia o tipo de valor que es agregado a un objeto *ArrayList* se convierte implícitamente a *Object*. Si los elementos son tipos de valor, se les debe aplicar la conversión *boxing* cuando se agregan a la lista y la conversión *unboxing* cuando se recuperan a su tipo específico. Tanto las operaciones de conversión de tipos como las conversiones *boxing* y *unboxing* requieren de muchos recursos; el

efecto de las conversiones *boxing* y *unboxing* resulta un factor importante en procesos donde se deben recorrer colecciones extensas.

Otra limitante es que no existe una comprobación de tipos en tiempo de compilación; puesto que un objeto *ArrayList* acepta tipos *Object*, todo se convierte a *Object* no existe en tiempo de compilación forma alguna de evitar que se realice lo siguiente:

```
System.Collections.ArrayList lista = new System.Collections.ArrayList();
// Agregar un entero a la lista.
lista.Add(3);
// Agregar una cadena a la lista. Se compilará, pero posteriormente se lanzará un error.
lista.Add("Mi cadena.");
int t = 0;
// Esto provoca una excepción del tipo InvalidCastException.
foreach (int x in lista) { t += x; }
```

Aunque parece válido si se crea una colección heterogénea, es probable que la combinación de cadenas y valores *int* en un objeto *ArrayList* único sea un error de programación, el cual no se detectará hasta el tiempo de ejecución. En las versiones 1.0 y 1.1 del lenguaje *C#*, se pueden evitar los riesgos de utilizar código generalizado en las clases de colección de la biblioteca de clases base de *.NET Framework* escribiendo colecciones propias específicas del tipo. Pero dicha clase no se puede reutilizar para más de un tipo de datos, por lo que se pierden las ventajas de la generalización (que los tipos sean homólogos) y se debe volver a escribir la clase para cada uno de los tipos que se almacenarán.

En los tipos genéricos se declara o se hace uso del tipo de datos en particular que se va a utilizar. Esto elimina el pesado proceso de convertir a *System.Object* y también hace posible realizar una comprobación de tipos por parte del compilador. En la colección genérica *List<T>*, en el espacio de nombres *System.Collections.Generic*, se tiene un método para agregar elementos a la colección, la cuál puede observarse en el siguiente fragmento de código:

```
// se crear una lista haciendo uso de una lista genérica de .NET Framework 2.0
List<int> lista1 = new List<int>();
// Sin operaciones de boxing, y sin casting (conversiones explicitas):
lista1.Add(3);
// Se genera un error en tiempo de compilación:
lista1.Add("Mi cadena.");
```

1.6.9.2 Interfaz IEnumerable<T>.

La interfaz genérica *IEnumerable<T>* (espacio de nombres *System.Collections.Generic*) fue incorporada en esta versión con el objetivo de trabajar en conjunto con los tipos genéricos y tiene el mismo funcionamiento que su homónima no genérica *IEnumerable* en *.NET 1.1* (*System.Collections*). Esta interfaz sirve como un mecanismo de iteración sobre los elementos de una secuencia dada y en la que se puede aplicar la sentencia de iteración *foreach*. En el siguiente fragmento de código en *C#* se puede observar la definición de esta interfaz:

```
// System.Collections.Generic
public interface IEnumerable<T> : IEnumerable{
        IEnumerator<T> GetEnumerator();
}
// System.Collections
public interface IEnumerable{
        IEnumerator GetEnumerator();
}
```

Esta interfaz incluye un único método *GetEnumerator()*, que, como se puede observar, devuelve un enumerador, un objeto que genera secuencialmente elementos del tipo *T*. Como se observa, *IEnumerable<T>* hereda de otra interfaz, *IEnumerable* y no es genérica por lo que debe implementar también una versión no genérica de *GetEnumerator()*, para la que generalmente sirve el mismo código de la versión genérica, esto con el fin de realizar el recorrido de colecciones genéricas desde código no genérico. A su vez, la interfaz *IEnumerator<T>* se encuentra definida de la siguiente forma en *C#*:

```
// System.Collections.Generic
public interface IEnumerator<T> : IDisposable, IEnumerator
{
    T Current { get; }
}

// System.Collections
public interface IEnumerator {
    object Current { get; }
    void Reset();
    bool MoveNext();
}
```

Como se puede notar, esta interfaz hace uso de su homónima no genérica. Por lo que *Enumerator<T>* tiene que implementar los siguientes miembros:

- La propiedad *Current*. Esta propiedad es la que devuelve el elemento actual de un tipo específico de la secuencia.
- El método *Reset()*. Este método reinicia la enumeración a un valor inicial.
- El método *MoveNext()*. Este método desplaza al enumerador al siguiente elemento de la secuencia. Devuelve false cuando se llega al final de la secuencia.
- El método *Dispose()*. Este método libera cualquier recurso no administrado asociado al enumerador, este método proviene de la implementación de la interfaz *IDisposable* pero que no se muestra en el fragmento mostrado arriba.

La clase que implemente *IEnumerator<T>* mantiene un estado necesario para garantizar que los métodos de la interfaz funcionen correctamente. De aquí se observa que *IEnumerable<T>* se encuentra por encima de *IEnumerator<T>,* y esta se encarga de realizar el proceso de enumeración. Esto permite una ejecución de iteraciones anidadas sobre una misma secuencia. Si la secuencia implementara directamente la interfaz enumeradora, solo se dispondría de una iteración en cada momento y en consecuencia no se podrían realizar iteraciones anidadas sobre una misma secuencia. En vez de eso, las secuencias implementan *IEnumerable<T>*, cuyo método *GetEnumerator()* genera un nuevo objeto de enumeración cada vez que es llamado.

Cuando se aplica la sentencia *foreach* a objetos que implementan *IEnumerable<T>* básicamente se sigue el siguiente patrón: se obtiene un enumerador llamando a *GetEnumerator(),* posteriormente se recorre de principio a fin a través del método *MoveNext()*, en el cuál dependiendo del valor devuelto (true si se ha llegado al limite de la secuencia, false en caso contrario) se terminará la iteración.

1.6.9.3 Tipos de valor Anulable.

Los tipos que aceptan valores *null* son instancias de la estructura *System.Nullable*. Un tipo que acepta valores *null* puede representar el rango normal de valores de su tipo de valor correspondiente, además del valor *null*. Por ejemplo, a un tipo *Nullable<Int32>,* se le puede asignar cualquier valor válido para un Int32 o el valor *null*, a un tipo *Nullable<bool>*, se le puede asignar los valores true o false, o *null*. Esta forma de funcionamiento se basa directamente en el ámbito de trabajar con bases de datos y otros tipos de datos que contienen elementos que pueden no tener un valor asignado. Por ejemplo, un campo *int* de una base de datos puede almacenar tanto valores tanto enteros como nulos. La clase *Nullable* proporciona compatibilidad para la estructura *Nullable<T>*. La clase *Nullable* admite la obtención del tipo subyacente de un tipo que acepta valores *null* y las operaciones de comparación e igualdad en pares de tipos que aceptan valores *null* y cuyos tipos de valor no admiten las operaciones genéricas de comparación e igualdad.

Los tipos de valor anulable tienen las siguientes características:

- Los tipos que aceptan valores *null* representan variables de tipo de valor a las que se puede asignar el valor *null*. No se puede crear un tipo que acepta valores null basado en un tipo de referencia, esto resulta obvio, ya que los tipos de referencia ya admiten el valor *null*.
- Una forma de abreviar la sintaxis *System.Nullable* <*T*> es haciendo uso de la expresión *T?*, donde *T* es un tipo de valor. Las dos formas son válidas.
- Se puede asignar un valor a un tipo que acepte valores *null* de forma normal, por ejemplo, *int*? entero = 10; o *double*? doble= 4.108;
- Se puede usar el método *System.Nullable.GetValueOrDefault()* para devolver el valor asignado o el valor predeterminado del tipo si el valor es *null*, por ejemplo *int j = x.GetValueOrDefault()*.

Ejemplo de declaración de estos tipos en *C#*:

```
static void Main(string[] args){
    //declaración de un tipo int que acepta valores NULL,
    //aquí se incluye el espacio de nombres
    System.Nullable<int> enteroNullable = new System.Nullable<int>();
    //declaración de un tipo bool que acepta valores NULL
    System.Nullable<bool> boloNullable=new System.Nullable<bool>();
    //Agregando valores NULL
    enteroNullable = null;
    boolNullable = null;
    //Agregando valores correctos;
    enteroNullable = 1;
    boolNullable = true;
    //Provoca un error en tiempo de compilación
    int entero = null;
    bool bol = null;
}
```

1.6.10 .NET FRAMEWORK 3.0.

. *NET Framework 3.0* incluye un nuevo conjunto de elementos que son parte integral de los sistemas operativos *Windows Vista* y *Windows Server 2008*. Esta versión también se encuentra disponible para *Windows XP SP2* y *Windows Server 2003* como una descarga. No se incluyen grandes cambios en la arquitectura de esta versión, en lo que respecta al *CLR* ya que sigue utilizando el *CLR* de. *NET Framework 2.0*. A diferencia de las versiones principales mencionadas anteriormente, en esta versión no se agrega una versión de *.NET Compact Framework*. Esta versión de *.NET Framework* se compone de cuatro nuevos elementos:

1.6.10.1 Windows Presentation Foundation (WPF).

Windows Presentation Foundation (*WPF*) agrega un nuevo subconjunto de tipos a los ya existentes en *.NET Framework*, los cuáles se encuentran (en su mayoría) ubicados en el espacio de nombres de *System.Windows*. *WPF* brinda nuevas características y mejoras tanto en la programación como en el diseño de interfaces de usuario que están centradas en el desarrollo de aplicaciones cliente de *Windows*, con lo que se logra una mejor independencia con la plataforma de hardware. Para programar una aplicación de este tipo, se hace uso del código de lenguaje marcado (*XML*) y del código del lenguaje orientado a objetos, lo que proporciona a los programadores una experiencia familiar al modelo utilizado en *ASP.NET*. En *WPF* se utiliza el lenguaje marcado de aplicaciones extensible (*XAML*) para implementar la apariencia de una aplicación, y los lenguajes de programación (subyacentes o *code-behind*) para implementar su comportamiento. Esta separación entre la apariencia y el comportamiento aporta las ventajas siguientes:

- Se reducen los costos de programación y mantenimiento, al no estar el diseño de la apariencia muy relacionado con el código que determina su comportamiento.

- El trabajo en la programación se divide, ya que al mismo tiempo que los diseñadores implementan la apariencia de una aplicación, los programadores implementan su comportamiento.

1.6.10.*2* Windows Communication Foundation (WCF).

Windows Communication Foundation (*WCF*) es una tecnología que proporciona una serie de herramientas *.NET* para la creación y ejecución de sistemas orientados a la conexión. *WCF* proporciona una infraestructura de comunicaciones que se basa en la arquitectura de servicios web. Este modelo de servicio incluye una asignación extensible y flexible de mensajes para la implementación de servicios en lenguajes para la plataforma de *.NET* como *C#* o *Visual Basic*. *WCF* brinda métodos de serialización, además de proporcionar integración e interoperabilidad con sistemas distribuidos de *.NET Framework* que ya se encuentran en funcionamiento, como *Message Queue Server* (MSMQ), *COM+* y servicios *Web ASP.NET* entre las más importantes. *WCF* introduce un nuevo entorno de desarrollo diseñado para trabajar de forma óptima con aplicaciones que no estén diseñadas con el propio *WCF*, de ahí surgen dos elementos importantes con respecto a esa interoperabilidad, la primera de ellas recae en la interoperabilidad con otras plataformas y la segunda en la interoperabilidad con las tecnologías propias de Microsoft que diseñadas con anterioridad a *WCF*.

1.6.10.3 Windows Workflow Foundation (WF).

Windows Workflow Foundation es un conjunto de actividades que resultan de utilidad a los clientes para coordinar tanto a personas como a sistemas, esto a través de flujos de trabajo humanos o de sistema en sus aplicaciones escritas para los sistemas operativos *Windows Vista*, *Windows* XP y *Windows Server 2003*. *Windows Workflow Foundation* se encuentra integrado por un espacio de nombres propio, un motor de flujo de trabajo en proceso y diseñadores. *Windows Workflow Foundation* se puede utilizar en escenarios simples, como mostrar controles de interfaz de usuario a partir de información recabada e ingresada por el usuario, o escenarios complejos encontrados por empresas grandes, como procesamiento de pedidos y control de inventario. *Windows Workflow Foundation* se incluye con un modelo de programación, un motor de flujo de trabajo personalizable y re hospedable, y herramientas para generar rápidamente las aplicaciones habilitadas para el flujo de trabajo en *Windows*.

1.6.10.4 Windows CardSpace.

Windows CardSpace proporciona un ambiente adecuado en la autenticación de usuarios, ya que ayuda a los usuarios a administrar mejor sus identidades digitales y proporciona un nivel de seguridad confiable en la protección de los usuarios contra ataques a su identidad, como por ejemplo la suplantación o "*phising*". Para esto se habilita a los usuarios para la creación y manejo de sus tarjetas de información y en el acceso a estas. Del mismo modo que la información que representan los permisos de conducir, los pasaportes y las tarjetas de crédito, está garantizada por una organización, cada tarjeta de información representa datos que están firmados digitalmente por un proveedor, por uno mismo o por un tercero. Estas tarjetas son representaciones de datos que están garantizados por una parte determinada. Cada tarjeta se puede utilizar en diferentes situaciones, esto debido a que *CardSpace* organiza estas tarjetas de información y permite a las personas decidir si presentarlas en los lugares en las que son compatibles.

1.6.11 .NET Framework 3.5.

Esta versión fue publicada el 19 de Noviembre de 2007 y no se incluye dentro de *Windows Server 2008*. Así como en *.NET Framework 3.0*, esta versión utiliza al *CLR* de la versión 2.0. Además, ésta integrada el *SP1* de *.NET Framework 2.0*, *.NET Framework 2.0 SP2* (junto con *3.5*

SP1) y el *SP1* de *.NET Framework 3.0*. Esta versión agrega algunos métodos y propiedades a las clases del *BCL* en la versión 2.0 la cuál es requerida para acceder a las características propias de la versión 3.5 tales como *Language Integrated Query* (*LINQ*). Estos cambios no afectan a las aplicaciones escritas para en la versión 2.0. Además se agregan funcionalidades a los compiladores de *C# 3.0* y *VB 9.0* que son de vital importancia como las que se describirán en los temas siguientes.

1.6.11.1 Métodos de extensión.

Los métodos de extensión son métodos que se pueden agregar a los tipos ya existentes sin tener que crear un nuevo tipo que derive del original y posteriormente tener que volver a compilar o sin necesidad de modificar el tipo original. Básicamente estos métodos son métodos estáticos, pero se les llaman a través del código como si fuesen métodos de instancia al tipo que extienden, es decir, los métodos de extensión se definen como métodos estáticos pero se llaman utilizando la sintaxis de los métodos de instancia.

Un claro ejemplo de estos métodos de extensión son los operadores de consulta estándar que usa *LINQ*, ya que permiten realizar operaciones de consulta sobre los tipos *System.Collections.IEnumerable* y *System.Collections.Generic.IEnumerable<T>* existentes. Para hacer uso de estos métodos de extensión que funcionan como operadores de consulta, debe hacerse una referencia al espacio de nombres *System.Linq* con la palabra reservada *using*. Con esto, se podrá observar que cualquier tipo que implemente *IEnumerable<T>* (como los arreglos y las listas) tiene métodos de instancia como *GroupBy*, *OrderBy*, *Average*, etc. (se verán más adelante). En la definición de un método estático, hay que tomar en cuenta los siguiente: en la declaración del primer parámetro, se debe especificar en qué tipo actúa el método y va precedido del modificador *this*.

En el ejemplo siguiente se muestra un método de extensión definido para la clase *System.String*. Observe que se define dentro de una clase estática no anidada y no genérica:

```
namespace MetodosExtension{
    public static class MiExtension {
        public static int ConteoPalabras(this String cadena){
            //Split regresa un arreglo de cadenas que se obtienen apartir de
            //los caracteres que se coloquen como separadores,
            //con la opción StringSplitOptions.RemoveEmptyEntries no se
            //incluiran cadenas vacias en el arreglo,
            //con Length se devuelve el numero de elementos en el arreglo (cadenas)
             return cadena.Split( new char[] { ' ', '.','?' },
                    StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

El método de extensión *ConteoPalabras* se puede referenciar utilizando la sentencia *using MetodosExtension*, en el siguiente fragmento de código se puede observar cómo es su uso y llamada:

```
using MetodosExtension;
        ...
namespace Linq
{
    class Linq {
        static void Main(string[] args) {
          //se declara una variable del tipo string conteniendo
          //una cadena de texto
          string cadena = "Metodos de Extension,realizado";
          //observe que no se hace referencia a la clase
          //MiExtension, sino que el método de extensión
          //declarado en este es accedido como si fuera parte
          //de la clase String
          int i = cadena. ConteoPalabras();
        }
    }
}
```

En el código, el método de extensión se invoca con sintaxis de método de instancia. Sin embargo, el lenguaje intermedio (*IL* o *CIL*) generado por el compilador convierte el código en una llamada en el método estático. Los métodos de extensión no pueden tener acceso a las variables privadas del tipo que extienden.

1.6.11.2 Propiedades auto-implementadas.

Esta nueva característica en la definición de las propiedades, hace que la sintaxis resulte más concisa cuando no se requiere implementar ninguna función en los descriptores de acceso de la propiedad. Al declarar una propiedad como se muestra en el siguiente ejemplo, el compilador crea un campo de respaldo privado y anónimo al que solamente puede obtenerse acceso a través de los modificadores de acceso *get* y *set* de la propiedad. En el ejemplo siguiente en *C#*, se muestra una clase simple en la cual se han definido este tipo de propiedades:

```
public class Cliente {
    public Cliente() { }
    // Propiedades Auto-implementadas

    public string Nombre { get; set; }
    public int Edad { get; set; }
    //Propiedad de solo lectura
    public string Ciudad { get; private set; }
}
```

Al igual que en la declaración de las propiedades normales, las propiedades auto-implementadas deben declarar un modificador de acceso *get* y uno *set*. Para crear una propiedad *readonly* (solo lectura) auto-implementada, debe asignarse un modificador de acceso *set private*.

1.6.11.3 Inicializadores de objeto y de colección.

Los inicializadores de objeto permiten asignar valores a los campos o propiedades accesibles de un objeto en el momento de la creación sin tener que invocar explícitamente al constructor. En el ejemplo siguiente se muestra cómo utilizar un inicializador de objeto para un tipo *Cliente* haciendo uso de las propiedades que tiene esta clase.

```
public class Cliente {
    public Cliente() { }
 // Propiedades Auto-implementadas
    public string Nombre { get; set; }
    public int Edad { get; set; }
    public string Ciudad { get; set; }
}
public static void Metodo() {
// inicializador de objeto para un tipo Cliente
    Cliente cliente = new  Cliente
                      {Nombre="Fanny",Edad=30,Ciudad="Estado"};
}
```

1.6.11.3.1 Inicializadores de objeto con tipos anónimos.

Aunque los inicializadores de objeto son útiles para usarse en cualquier momento, son especialmente útiles en expresiones de consulta *LINQ.* Esto se debe a que en las expresiones de consulta a menudo se declaran y se hacen uso de tipos anónimos, los cuales sólo se pueden inicializar con un inicializador de objeto, esto debido a la naturaleza del objeto anónimo. En la cláusula *select*, en una expresión de consulta se puede transformar los objetos de la secuencia original (la que nos sirve como fuente de datos, es decir a la que se le aplicará una consulta) en objetos cuyo valor y forma pueden ser distintos de los originales. Esto es permite almacenar, si se quiere, una parte de la información en cada objeto de una secuencia. En el ejemplo siguiente se supone que un objeto de cliente (*clienteInfo*) contiene muchos campos y métodos, y el usuario

sólo está interesado en crear una secuencia de objetos que contenga el nombre del cliente y su edad.

```
//clientes es un elemento que implementa a IEnumerable
//y contiene varios objetos del tipo Cliente
var clienteInfo =
      from p in clientes
      select new { p.Nombre, p.Edad};
```

1.6.11.3.2 Inicializadores de objeto con tipos que aceptan valores null.

Es un error en tiempo de compilación utilizar un inicializador de colección con una estructura que acepta valores *null*.

1.6.11.3.3 Inicializadores de colección.

Estos inicializadores permiten especificar uno o varios inicializadores de elemento al declarar e inicializar una colección que implementa *IEnumerable*. Los inicializadores de elemento pueden ser un valor simple, una expresión o un inicializador de objeto. Si se utiliza un inicializador de colección, no es necesario especificar varias llamadas al método *Add* de la clase en el código fuente; el compilador agrega las llamadas. En el ejemplo siguiente se muestra los inicializadores de colección simples:

```
List<int> numeros = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> numeros2 = new List<int> { 0 + 1, 12 % 3, AlgunMetodoInt() };
```

El inicializador de colección del siguiente fragmento de código hace uso de inicializadores de objeto para inicializar los objetos de la clase *Cliente*, que se define en el ejemplo del apartado 1.2.11.2 y para declarar varios inicializadores de objeto individuales estos se separan por comas.

```
List<Cliente> c = new List<Cliente>
{
    new Cliente { Nombre="Luis", Edad=8,Ciudad="Saltillo" },
    new Cliente { Nombre="Martha", Edad=2, Ciudad="Nuev León"},
    new Cliente { Nombre="Brenda", Edad=22, Ciudad="DF"}
};
```

1.6.11.4 Interfaz IQueryable<T>.

Esta interfaz permite realizar consultas con a un origen de datos en específico en el que se conoce el tipo de los datos (espacio de nombres *System.Linq*). Donde *T* es el tipo de los datos del origen de datos. La interfaz *IQueryable<T>* está pensada para ser implementada por los proveedores de consultas.

```
public interface IQueryable<T> : IEnumerable<T>, IQueryable, IEnumerable
```

Esta interfaz se apoya de la interfaz *IEnumerable<T>,* por lo que se pueden enumerar los resultados de una consulta que se quiera realizar. La enumeración fuerza la ejecución del árbol de expresión asociado a un objeto *IQueryable<T>*, la cual es específica de un proveedor de consultas. La interfaz *IQueryable<T>* permite que las consultas sean polimórficas, es decir, como una consulta con respecto a un origen de datos *IQueryable* se representa como un árbol de expresión, se puede ejecutar con respecto a diferentes tipos de orígenes de datos.

1.6.11.4 Inferencia de tipos.

1.6.11.4.1 Tipo var implícito.

Las variables que se declaran dentro de un método pueden tener un tipo inferido (*var*) en lugar de un tipo explícito. A través de la palabra clave *var* se indica al compilador que infiera el tipo de la variable local declarada a partir de la expresión que se encuentra en el lado derecho de

su declaración. El tipo deducido puede ser un tipo integrado, un tipo anónimo, un tipo definido por el usuario o un tipo definido en la biblioteca de clases de *.NET Framework*.

```
var i = 10; //tipo implicito
int i = 10; // tipo explicito
```

En los ejemplos siguientes se muestran varias maneras de declarar variables locales con *var*:

```
// i es compilado como un int
var i = 5;

// s es compilado como un string
var s = "Cadena";

// a es compilado como un int[]
var a = new[] { 0, 1, 2 };

// query es compilado como un IEnumerable<T>
var query =
    from c in clientes
    where c.Ciudad == "Mexico"
    select c;

// anónimo es compilado como un tipo anónimo
var anonimo = new { Nombre = "Miguel", Edad = 20 };

// lista es compilada como un List<int>
var lista = new List<int>();
```

Es importante entender que la palabra clave *var* no significa "*variant*" ni indica que la variable sea de relación en tiempo de ejecución ni mucho menos *Object*. Simplemente significa que el compilador determina y asigna el tipo más adecuado.
La palabra clave *var* se puede usar en los siguientes contextos:

- En variables locales (variables declaradas dentro del método), como se mostraba en el ejemplo anterior.
- En una instrucción de inicialización *for*-> for(var x = 1; x < 10; x++)
- En una instrucción de inicialización *foreach*-> foreach(var item in list){...}
- En una instrucción *Using*-> using (var file = new StreamReader("C:\\miArchivo.txt")) {...}

1.6.11.4.2 Tipos anónimos.

Los tipos anónimos encapsulan un conjunto de propiedades de sólo lectura en un único objeto sin tener que definir antes un tipo de forma explícita. El compilador genera automáticamente el nombre de tipo y con esto el nombre no estará disponible en el código fuente que utiliza el programador. El compilador deduce el tipo de las propiedades. En el ejemplo siguiente se muestra un tipo anónimo que se inicializa con dos propiedades denominadas *Nombre, Edad y Ciudad*.

```
var v = new { Nombre = "Maria" Edad = 108, Ciudad="Mi ciudad"};
```

Un ejemplo del uso de estos tipos es en *LINQ*, ya que a menudo en la cláusula *select* de una consulta se devuelven un subconjunto de las propiedades de cada objeto de la secuencia de origen. Los tipos anónimos se crean utilizando el operador *new* con un inicializador de objeto. Los tipos anónimos son tipos clase que constan de una o más propiedades públicas de sólo lectura así que no se permite ningún otro tipo de miembros de clase, como los métodos o eventos. Un tipo anónimo no se puede convertir a ninguna interfaz ni tipo solo a *object*. En el siguiente ejemplo se utiliza una clase denominada *Cliente* que incluye las propiedades *Nombre, Edad y Ciudad*. "*clientes*" es una colección genérica de objetos *Cliente*. La declaración de tipo anónimo comienza por la palabra clave *new*. Inicializa un nuevo tipo que utiliza dos de las propiedades de la clase *Cliente*. Esto hace que la consulta devuelva una cantidad de datos menor.

Si no se especifican los nombres de miembro para el tipo anónimo, el compilador asigna a los miembros de ese tipo anónimo el mismo nombre que la propiedad que se utilice para inicializarlos. Se debe proporcionar un nombre a una propiedad que se inicialice con una expresión:

```
namespace Ejemplos
{
    //Clase Cliente
    public class Cliente {
        public Cliente() { }
        //Hacendo uso de propiedades auto-implementadas
        public string Nombre { get; set; }
        public int Edad { get; set; }
        public string Ciudad { get; set; }
    }

    class Program {
        static void Main(string[] args) {
            //realizando instancias de la clase Cliente
            Cliente c1= new Cliente {Nombre="Luis",Edad=30,Ciudad="Paris"};
            Cliente c2= new Cliente {Nombre="Miguel",Edad=20,Ciudad="Aguascalientes"};
            Cliente c3= new Cliente {Nombre="Adriana",Edad=24,Ciudad="Torreon"};
            Cliente c4= new Cliente {Nombre="Brenda",Edad=30,Ciudad="Df"};
            //tipo anonimo que se inicializa con solo 2 valores que tiene Cliente
            var v = new { Nombre = "Maria", Edad = 108 };
            //agregando a la lista generica clientes todos los elementos
            //para que posteriormente esta lista (que implementa
            //la interfaz IEnumerable<T> sirva como una fuente de consulta)
            List<Cliente> clientes= new List<Cliente>(){c1,c2,c3,c4};
            //realizando la consulta SQL, notese que se ha utilizado
             //inferencia de tipo usando var para variable inferida query
            var query = from q in clientes
              //asignando valores a c/u de las propiedades que se van a seleccionar,
             //si no se hace el compilador los asigna automáticamente de forma secuencial
                        select new { valor1 = q.Nombre, valor2 = q.Edad};
            //En la expresion anterior solo se han obtenido 2 de las 3
            //propiedades de la clase Cliente (Nombre,Edad) por lo que
            //solamente estarán disponibles estas 2 propiedades para query
            foreach(var cliente in query)
             Console.WriteLine(cliente.valor1 + cliente.valor1) ;
        }
    }
```

1.6.11.5 Expresiones lambda.

Una expresión lambda es un método anónimo que realiza expresiones o instrucciones y se puede usar para crear delegados o árboles de expresión.

En una expresión lambda se usa el operador =>, en el cual del lado izquierdo este operador se colocan los parámetros de entrada (si existe alguno), mientras que en el lado derecho se declara la expresión o instrucción.

La manera natural de implementar una definición de una función lambda que calcula el cuadrado de un número en *C#* sería a través de un tipo y una instancia de delegados:

```
//definiendo al delegado genérico Delegado<T> que regresará un tipo T
//declarado después de la palabra reservada delegate, es decir dependiendo
//del tipo ingresado se devolverá el mismo tipo
public delegate  T Delegado<T>(T x);
//A partir del modelo anterior se crea una instancia, que apunta a un
//método anónimo (delegate) que regresa la variable multiplicada por sí mismo
Delegado<int> cuadrado = delegate(int x) { return x * x; };
```

Del fragmento de código mostrado arriba se observan 2 cosas importantes:

- En la primera línea del código anterior se define un delegado genérico llamado *Delegado<T>*. A partir de este delegado se puede crear instancias para métodos que a partir de un valor tipo T regresan otro valor del mismo tipo.
- En la segunda línea se define una instancia del delegado anterior que apunta a un método que devuelve el cuadrado de un número entero. En esta sintaxis se hace uso de otra característica incorporada a *C#* en la versión 2.0, los métodos anónimos (por cuestiones de espacio no se vieron puesto que las expresiones lambda simplifican este concepto), que permiten la definición en línea de una instrucción que especifica la funcionalidad a la que se desea asociar a la instancia del delegado.

En el siguiente fragmento de código se muestra la implementación a través de código:

```
namespace Linq{
    class Linq{
        public delegate  T Delegado<T>(T x);
        static void Main(string[] args){
            Delegado<int> cuadrado = delegate(int x) { return x * x; };
            //El resultado a mostrar sería 4
            Console.WriteLine(cuadrado(2));
        }
   }
}
```

Las expresiones lambda resultan de vital importancia en los métodos anónimos, ya que ofrecen una sintaxis más concisa y funcional para realizar una consulta. Por ejemplo, la definición del cuadrado mostrado anteriormente queda de la siguiente forma utilizando una expresión lambda:

```
Delegado<int> cuadrado = x => x * x;
```

Y su implementación de la siguiente:

```
namespace Linq{
    class Linq{
        public delegate  T Delegado<T>(T x);
        static void Main(string[] args){
            //Haciendo uso de una eexpresion lambda
            Delegado<int> cuadrado2 = x => x * x;
            //El resultado a mostrar sería 4
            Console.WriteLine(cuadrado(2));
        }
   }
}
```

En la mayor parte de las veces se usan las distintas sobrecargas del tipo genérico predefinido llamado *Func*, la cual se encuentra en el ensamblado *System.Core.dll* de *.NET 3.0* y en el espacio de nombres *System*:

```
public delegate T Func<T>();
public delegate T Func<A0, T>(A0 a0);
public delegate T Func<A0, A1, T>(A0 a0, A1 a1);
public delegate T Func<A0, A1, A2, T>(A0 a0, A1 a1, A2 a2);
public delegate T Func<A0, A1, A2, A3, T>(A0 a0, A1 a1, A2 a2, A3 a3);
```

El tipo *Func* representa a los delegados a funciones (con 0, 1, 2 ó 3 argumentos, respectivamente) que devuelven un valor de tipo *T*. Utilizando una de estos delegados, se puede redefinir el método del cuadrado de la siguiente forma:

```
Func<int, int> cuadrado = x => x * x;
```

La utilidad de las expresiones lambda en LINQ radica esencialmente sobre las consultas y específicamente en los métodos, ya que se usan como argumentos para estos últimos, dando como resultado una expresión a evaluar por parte de los operadores de consulta estándar, tales como *Where* (el cuál tiene como parámetro el tipo *Func* mencionado anteriormente). Existen dos tipos de expresiones lambda: una expresión lambda con una expresión en el lado derecho se denomina

una lambda de expresión (también dicho expresión lamba), el segundo tipo es una lambda de sentencia (también sentencia lambda), que es similar a una lambda de expresión excepto que en la parte derecha consiste en una serie de sentencias encerradas en llaves.
Para dar una mejor idea sobre las expresiones lambda se muestra la siguiente tabla usando *C#*:

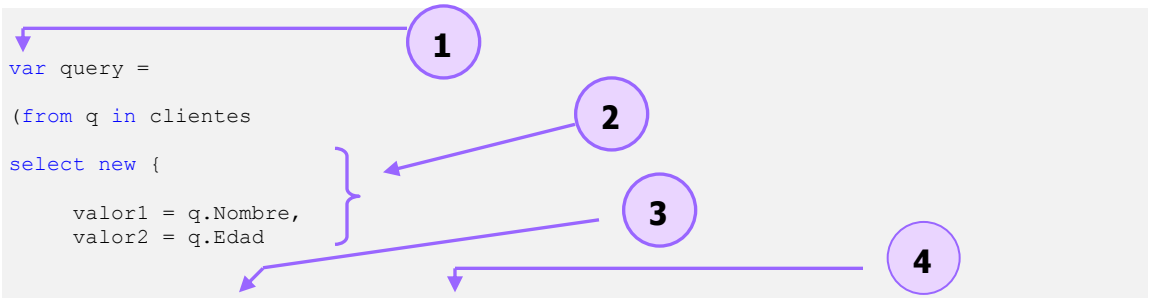
```
1.  x => x + 1
2.  x => { return x + 1; }
3.  (int x) => x + 1
4.  (int x) => { return x + 1; }
5.  (x, y) => x * y
6.  () => 1
7.  () => Console.WriteLine()
    cliente => cliente.Nombre
    persona => persona.Cuidad == "Paris"
    (persona, minEdad) => persona.Edad >= minEdad
```

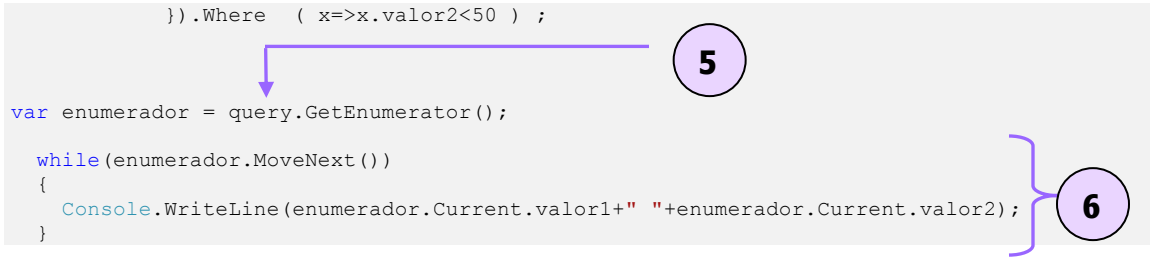
Donde se tiene que es:
1. Implícitamente tipada (*int*), lamba de expresión.
2. Implícitamente tipada (*int*), lamba de sentencia.
3. Explícitamente tipada (*int*), lamba de expresión.
4. Implícitamente tipada (*int*), lamba de sentencia.
5. Múltiples parámetros.
6. Sin parámetros, lamba de expresión.
7. Sin parámetros, lamba de sentencia.

1.7 COMO LINQ EXTIENDE A .NET PARA REALIZAR CONSULTAS SQL EN EL LENGUAJE.

En esta sección se pone un panorama claro para que se pueda tener una idea clara de cómo los elementos de *.NET Framework* presentados anteriormente trabajan en conjunto cuando se utilizan con *LINQ*. Todos estos elementos y otros que no se mencionaron par falta de espacio, los cuales están presentes en las diversas versiones de *.NET Framework*, pueden denominarse extensiones al lenguaje (en este caso *C#* y *VB.NET*, aunque como se mencionó al principio, por razones de espacio solo se hizo uso de *C#*), un conjunto tanto de características sintácticas en versiones como la 3.5 de *.NET* como las que se encuentran en versiones pasadas a esta, las cuáles dan un soporte firme para implementar y hacer uso de *LINQ*. Estas características, pueden ser usadas por sí solas sin tener que hacer uso de *LINQ*, sin embargo, son necesarios para que funcione *LINQ* y que le permitirán al programador realizar consultas integradas en el lenguaje, consultas *SQL* de forma nativa.

Para ello, se muestra el siguiente fragmento de código con el que se expone que lo presentado anteriormente no fue en vano y que tales constructos son importantes. Para poder desarrollar un claro ejemplo sobre una consulta , supóngase que se tiene una variable válida llamada clientes la cuál es una lista genérica de tipo *T*, que incluye 3 miembros públicos asignados cada uno a propiedades cuyo nombre son: *Nombre* (*string*), *Edad* (*int*) y *Ciudad*(*string*), sin métodos. Se tiene la siguiente operación de consulta SQL en LINQ y se muestran los resultados a través de consola:





Del código presentado anteriormente y retomado los elementos presentados en las versiones de *.NET Framework* se tiene lo siguiente:

1. Se declara una variable local de tipo inferido (*var*): *query*.
2. Se declara un tipo anónimo (‘*a*) cuyo resultado será pasado a la variable *query*. En este caso solo se obtienen 2 propiedades. A su vez se hace uso de los inicializadores de objetos.
3. Se hace uso de un método de extensión, en este caso *Where* (recordar que estos agregan funcionalidades a las clases para las que son definidas, en este caso *Where* es un método estático proveniente de la clase *Enumerable* en el espacio de nombres *System.Linq*, que nos sirve para poder realizar consultas a objetos que implementen la interfaz *IEnumerable<T>*, se mostrará en el siguiente capitulo) para poder aplicar un filtro de selección a la secuencia de valores contenida en clientes. En este caso la sobrecarga que se usa es la de *Func<A0, T>(A0 a0)* (el parámetro del método), que como se mencionó anteriormente, apunta hacia una función predefinida para el método *Where*.
4. Se tiene una expresión lambda: un predicado de condición el cuál será aplicado a cada uno de los elementos para que lo cumplan. En este caso se tendría:
   - *Func<A0,T>(A0 a0)*, el valor devuelto por *T* es del tipo *bool* al hacerse la comparación del *tipo* *x=>x.valor2<50* y el parámetro de entrada es un *IEnumerabel<T>*, donde *T* es anónimo (‘*a*), es decir *Fun<IEnumerable<’a>,bool>*
5. El resultado es una variable del tipo *IEnumerable<T>*, donde *T* pasa a ser el tipo anónimo que se declaró en el paso 2. En este caso, para poder observar cómo es que se usa la interfaz *IEnumerable<T>*, se procede a usar el método *GetEnumerator()*, el cuál como se mencionó anteriormente, se encuentra en la interfaz *IEnumerable <T>* y puesto que la variable *query* es de este tipo, este método es accesible. Este método nos regresará al enumerador para poder iterar a través de la colección.
6. Se realiza un ciclo haciendo uso de la estructura de control *while* y del método *MoveNext()* de la variable inferida enumerador, la cuál es del tipo *IEnumerator<T>*, y *T* también es del tipo anónimo. Mientras *MoveNext()* no nos devuelva false, se obtendrá la propiedad *Current*, la cual hace referencia a esta variable anónima, es decir que nos devuelve el objeto anónimo, y como se ve en el paso 2, solo se declararon 2 propiedades o valores, los cuales son visibles al programador.

Como se ha mencionado previos temas, el compilador traduce esta expresión a una serie de llamadas a métodos de extensión, con lo cual el resultado dado por el compilador de *C#* sería el siguiente:

```
System.Collections.Generic.List<Cliente> .Select ( q =>  new  {
                    valor1 = q.Nombre,
                    valor2 = q.Edad }
) .Where (x => (x.valor2 < 50))
```

CAPITULO II. CONCEPTOS SOBRE EJECUCIÓN DE SENTENCIAS SQL DE LINQ EN FORMA NATIVA PARA EL LENGUAJE

2.1 ENSAMBLADOS PROVISTOS EN .NET FRAMEWORK 3.5 PARA LINQ.

En la siguiente tabla presentada al inicio de esta tesis, se muestran los espacios de nombres y ensamblados disponibles para *LINQ* en la versión 3.5 de *.NET Framework:*

.NET FrameWork 3.5		
Tecnología	**Ensamblado**	**Espacio de nombres**
LINQ to Objects	System.Core.dll	System.Linq
LINQ to XML	System.Xml.Linq.dll	System.Xml.Linq
LINQ to SQL	System.Data.Linq.dll	System.Data.Linq
LINQ to Entities	System.Data.Entity.dll	System.Data.Objects y otros

Dado que estos espacios de nombre contienen un número extenso de clases, sería muy tedioso exponerlos todos, por esta razón en la tabla anterior solo se mencionan los espacios de nombres, sin embargo hay que recordar que dan soporte a cada una de las extensiones de *LINQ*.

2.2 ¿QUÉ ES UNA EXPRESIÓN DE CONSULTA?.

Los métodos de extensión o el uso de las nuevas palabras agregadas al lenguaje de *C# 3.0* y *VB 9.0* ofrecen sintaxis diferentes para realizar una consulta sobre una fuente de datos. Hay ocasiones en que una consulta *LINQ* exige el uso de la palabra reservada *var*. Esto es cuando se proyecta un tipo anónimo como se muestra en el siguiente ejemplo:

```
//Donde clientes es una lista genérica del tipo anónimo (List<anónimo>)
     var query = clientes
                 .Where(q => (q.Edad < 50))
                 .Select(q => new { valor1 = q.Nombre, valor2 = q.Edad });
```

Nota: en este y los subsecuentes ejemplos se hará uso de la clase Cliente y declarada en el capitulo anterior, en el apartado 1.2.11.2 (Propiedades auto-implementadas).

Pero otra forma de realizar esta operación sería de la siguiente forma, la cuál puede denominarse como una expresión de consulta. Una expresión de consulta ofrece una sintaxis similar a una sentencia *SQL*, por ejemplo:

```
//Donde clientes es una lista genérica del tipo anónimo (List<anónimo>)
     var query =from q in clientes
                where q.Edad< 50
                select new{valor1 = q.Nombre,valor2 = q.Edad} ;
```

Con esto se ofrece al programador una sintaxis integrada en el lenguaje lo que deriva en un uso más eficaz de los operadores de consulta, como lo hace un lenguaje de consulta relacional como lo es SQL. Básicamente existen 3 formas de realizar una consulta sobre una fuente de datos en *LINQ*:

1. A través de una expresión de consulta.
2. A través de las llamadas a métodos en caso de que un operador de consulta no tenga una palabra reservada (cláusula) en el lenguaje (se verá más adelante en los operadores de consulta estándar). Las llamadas a métodos de consulta se pueden encadenar juntas en una sola consulta, lo cual permite hacer consultas arbitrariamente complejas y de la cuál surgen 2 subtipos:

- Invocación de un método como si se tratara de un método de instancia a través del uso directo de la secuencia de elementos que implementan a *IEnumerable<T>*.
- Invocación de un método como si se tratara de un método estático a través de la clase estática *Enumerable* ó *Queryable*.

3. Una combinación de las dos primeras, es decir en una expresión de consulta hacer uso explicito de un método de extensión.

2.3 ANALIZANDO LA PRIMERA EXPRESIÓN DE CONSULTA.

Una expresión de consulta lleva a cabo operaciones sobre el contenido de las fuentes proporcionadas (colecciones de objetos) apoyándose en los métodos de extensión y son una parte principal de *LINQ*. Para poder observar cómo es que funcionan estos métodos desde su forma interna de una forma rápida tomaremos como ejemplo el método *Where* del siguiente fragmento de código:

```
var query =from q
       in clientes
       where q.Edad < 50
       select new{
         valor1=q.Nombre,
         valor2 = q.Edad
       } ;
```

```
var query = clientes
          .Where(q => (q.Edad < 50))

          .Select(q => new {
               valor1 = q.Nombre, valor2 = q.Edad
           });
```

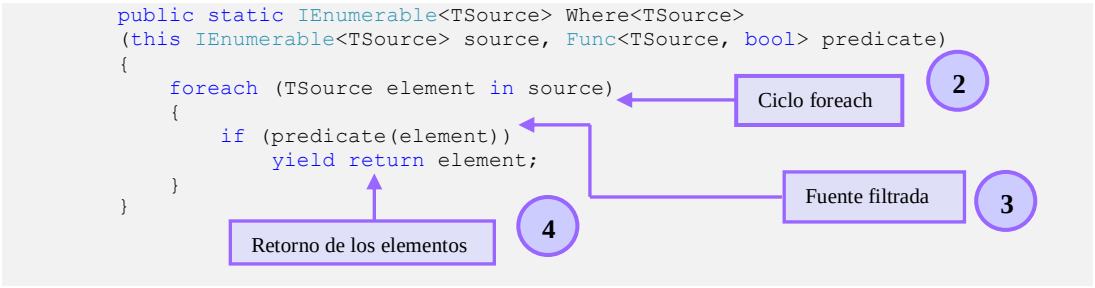
1 Llamada al método Where

Este método y Los operadores de consulta estándar se encuentran en el ensamblado System.Core.dll dentro del espacio de nombres *System.Linq* de *.NET Framework 3.5* como métodos de extensión en las clases estáticas *Enumerable* y *Queryable*. Pueden usarse en objetos que implementan *IEnumerable<T>* o *IQueryable<T>*. Esto les permite operar con varios tipos, desde colecciones y arreglos (secuencias) en memoria hasta bases de datos remotas que usan proveedores como *LINQ to Entities* y *LINQ to SQL*, es por eso que al hacer uso de las colecciones que implementan esta interfaz, se pueden utilizar estos métodos:

```
public static class Enumerable{
  //Metodos estaticos que se encuentran definidos en esta clase
  public static TSource Aggregate<TSource>(this IEnumerable<TSource> source,
                                           Func<TSource, TSource, TSource> func);
  ...
    //Sobrecarga 1 del método Where
  public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
                                                    Func<TSource, bool> predicate);

    //Sobrecarga 2 del método Where
  public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
                                                    Func<TSource, int, bool> predicate);
  ...
}
```

Como se puede observar, en el primer parámetro se especifica en qué tipo actúa el método y va precedido del modificador *this*, en este caso se trata de la fuente a la cuál se va a consultar (clientes), la cuál es una lista genérica y esta lista genérica implementa la interfaz *IEnumerable<T> (this IEnumerable<TSource> source)*. El segundo parámetro es una expresión lambda, la cuál varía para cada una de las sobrecargas. En el primer parámetro (para ambas sobrecargas de este método) se especifica el tipo de la fuente. La segunda en el caso de la primera sobrecarga y la tercera para la segunda sobrecarga del método, es el valor a devolver por esta expresión lambda, la cual es del tipo *bool*. Dado que el valor devuelto es de este tipo, la expresión lambda debe hacer este tipo de comparación. Ahora observemos más de cerca a la primera sobrecarga de este método que se encuentra implementado en esta clase:

```
public static IEnumerable<TSource> Where<TSource>
(this IEnumerable<TSource> source, Func<TSource, bool> predicate)
{
    foreach (TSource element in source)
    {
        if (predicate(element))
            yield return element;
    }
}
```

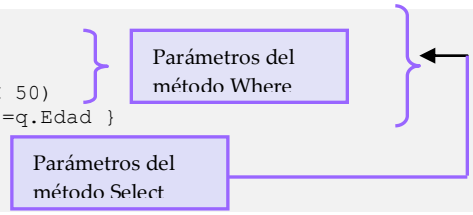


Aplicando el ejemplo que incluye a la clase Cliente en este caso el método toma un *IEnumerable<Cliente>* como argumento y regresa también un *IEnumerable<Cliente>*. Por lo que el método trabaja de la siguiente forma:

1. Este es llamado a través de la lista clientes.
2. Se itera sobre la lista clientes que es recibida.
3. Se filtra la lista clientes usando la expresión lambda dada como único parámetro, ya que al llamarse solo aparecerá este parámetro.
4. Si el elemento cumple con la expresión lambda, se regresa un elemento a la vez, esto gracias a los iteradores, en el cuál en vez de regresar una colección conteniendo todos los elementos ya filtrados se regresa un solo elemento a la vez. Para esto se hace uso de la sentencia *yield return*. Gracias a esto, cuando se alcanza esta sentencia el elemento que cumpla con la condición de la expresión lambda es retornado hacia la variable *query*, pero solo un elemento, lo que significa que el control es devuelto al la variable *query* para posteriormente pasar de nuevo al método en el que se está iterando, en este caso *Where* y con esto se pasa al siguiente elemento de la lista en donde se había quedado, para realizar la misma tarea hasta alcanzar el final de la lista clientes. Es interesante mencionar que el valor de retorno del método es un *IEnumerable<Cliente>*, pero al observar el código parece que no es así, solo se regresa un elemento, es decir al parecer se está regresando un solo objeto al hacer uso de *yield return*. Pero no es así, ya que el compilador genera automáticamente en su lenguaje intermedio una clase que implemente este tipo (*IEnumerable<Cliente>*), con lo que se genera un "motor de estado" en lenguaje intermedio, con el que se mantiene el estado del método al devolverse el valor a la variable *query* y volverse a iniciar a partir de ese estado en el que se había quedado.

Así como funciona este método algunos otros como *Select* también lo hacen, aceptan un *IEnumerable<T>* y también devuelve un *IEnumerable<T>*, por lo que el patrón descrito anteriormente aplica para este y otros métodos de extensión. Los operadores de consulta estándar difieren en el momento de su ejecución, dependiendo de si devuelven un valor *singleton* (un solo valor) o una secuencia de valores. Los métodos que devuelven un valor *singleton* (por ejemplo, *Average* y *Sum*) se ejecutan inmediatamente. Los métodos que devuelven una secuencia retrasan la ejecución de la consulta y devuelven un objeto *enumerable* (en este caso *Where*). Como ya se ha mencionado que los operadores de consulta residen en la clase *Enumerable* como métodos de extensión estáticos, entonces una consulta se puede realizar también como una secuencia de llamadas a métodos estáticos haciendo uso directo de la clase *Enumerable*, es decir que la expresión mostrada anteriormente queda de la siguiente forma:

```
var query =Enumerable.Select(
              Enumerable.Where(
                        clientes,
                        cliente => cliente.Edad < 50)
              , q => new { valor1=q.Nombre, valor2=q.Edad }
          );
```



Para mostrar los resultados y compararlos con la expresión de consulta se puede hacer uso del siguiente fragmento de código:

```
foreach (var item in query) { Console.WriteLine(item.valor1 + item.valor2.ToString()); }
```

2.3.1 EJECUCIÓN DE CONSULTAS.

2.3.1.1 Consulta.

Una consulta es una expresión que se evalúa en contra de un origen de datos y posteriormente devuelve los resultados contenidos en datos. Las consultas se desarrollan y expresan utilizando una sintaxis que corresponde a un lenguaje de consulta especializado, como lo es *SQL* para bases de datos. Cuando se realiza una operación de consulta en *LINQ* se básicamente se contemplan tres acciones: obtener el origen o los orígenes de datos, crear la consulta y ejecutar esa consulta.

Como se ha visto hasta ahora, en *LINQ* una consulta es asignada a una variable, con lo que el resultado es almacenado a esta variable. Si el resultado es una secuencia de valores, esta variable debe ser de un tipo que se pueda consultar. Esta variable de consulta no realiza ninguna acción y no devuelve datos; solamente almacena la información de la consulta. Tras crear la consulta (escribiendo la expresión), esta debe ejecutarse para recuperar la información. En una consulta que devuelve una secuencia de valores, la variable de consulta por sí misma nunca conserva los resultados de la consulta y sólo almacena sus órdenes o lo que tiene que realizar.

2.3.1.2 Ejecución diferida.

Este tipo se da cuando la ejecución de la consulta se produce en el momento en que se enumera el resultado, no en el instante de ser creada, esto es por qué la sentencia cuando es definida dentro del objeto *query*, ésta todavía no ha sido ejecutada, es decir, todavía no se han devuelto los resultados. Este objeto almacenado y que es asignado a *query* ya implementa la interfaz *IEnumerable<T>*. Así que hasta este punto *query* sólo sabe que tiene que hacer, pero todavía no ha hecho nada. La consulta será ejecuta cuando ésta sea enumerada a través del método *GetEnumerator*, el cual hace uso del operador *yield*, por lo que antes de iterar sobre la secuencia a través de la sintaxis de *foreach* la consulta no se ha ejecutada realmente.

```
//Construyendo la consulta
var query =from q in clientes where q.Edad < 50
           select new{valor1=q.Nombre,valor2 = q.Edad} ;
//Ejecutando la consulta
foreach(var item in query)Console.WriteLine(item);
```

2.3.1.3 Ejecución inmediata.

Esta situación sucede cuando ésta la consulta es ejecutada en el momento de su declaración. En este caso, para poder realizar una ejecución inmediata se tienen operadores como: *ToArray*, *ToList*, *ToLookup*, *ToDictionary*, etc. Estos operadores lo que hacen es cambiar la estructura de datos devuelta por la consulta y poner en memoria el resultado.

```
//Construyendo y ejecutando la consulta
var query =(from q in clientes where q.Edad < 50
            select new{valor1=q.Nombre,valor2 = q.Edad}).ToList() ;
```

2.4 OPERADORES DE CONSULTA ESTÁNDAR DE LINQ.

Esencialmente, los operadores de consulta estándar que componen a *LINQ* son métodos. De forma predeterminada e independientemente de las extensiones que derivan de *LINQ*, por si mismo es muy eficaz. Al hacer uso de estos operadores de consulta estándar con *LINQ*, se proporciona al programador un control más exacto y versátil sobre un conjunto de datos. Estos operadores le dan la capacidad al programador proporcionan de realizar operaciones de consulta

en el lenguaje nativo de *.NET* como filtración de datos, proyección de estos datos, agregación y ordenamiento por mencionar algunos. Existen 2 conjuntos de operadores que trabajan sobre secuencias de objetos diferentes: uno que funciona sobre secuencias del tipo *IEnumerable<T>* y otra sobre secuencias del tipo *IQueryable<T>,* cuyos métodos son miembros estáticos de las clases *Enumerable* y *Queryable*, respectivamente.

Sin embargo, es de vital importancia mencionar primero que no todos los métodos que corresponden a estos operadores tienen su equivalencia en una expresión de consulta a través de la sintaxis del lenguaje. En los ejemplos anteriores los principales métodos como *where*, *select* tienen una palabra clave asignada (cláusula) en el lenguaje de programación pero no sucede lo mismo con otros métodos, para poder aclarar esto, se abarcarán los operadores que se presentaron anteriormente y se dará la correspondiente sintaxis de la expresión de consulta en *C# 3.0*, si es que se tiene soporte. En los ejemplos subsecuentes, se mostrarán diferentes formas de realizar una consulta (a través de una expresión de consulta, a veces haciendo uso de métodos de extensión como métodos de instancia, una combinación, etc), esto no significa que se deben de realizar tal cuál están escritas, como se mencionó anteriormente, una consulta en *LINQ* se puede escribir de 3 formas distintas, la elección depende del programador.

Para mostrar y ejemplificar algunos de estos operadores a través de código para que resulte más entendible, a lo largo de este tema se tomará las siguientes clases para algunos de los ejemplos:

```
public class Producto{
    public string serieProducto { get; set; }
    public string nombre { get; set; }
    public decimal precio { get; set; }
    public string sucursal { get; set; }
}
public class Cliente{
    public int idCliente { get; set; }
    public string nombre { get; set; }
    public string direccion { get; set; }
    public int edad { get; set; }
}
public class Orden{
    public int idOrden { get; set; }
    public int idCliente { get; set; }
    public string serieProducto { get; set; }
}
```

Los cuales tendrán los siguientes valores iniciales con el siguiente fragmento de código:

```
Producto p1 = new Producto() { serieProducto = "AXCM3900908", nombre = "Laptop", precio =
600.00M, sucursal = "Techno Micro" };
Producto p2 = new Producto() { serieProducto = "ZWCE0000908", nombre = "Camara", precio =
999.99M,sucursal= "Video S.A." };
Producto p3 = new Producto() { serieProducto = "WASM3901208", nombre = "Disco DVD", precio
= 19.00M,sucursal= "Video S.A."};
Producto p4 = new Producto() { serieProducto = "MKAN1270908", nombre = "Lampara", precio =
259.99M, sucursal = "Techno Micro" };
Producto p5 = new Producto() { serieProducto = "WXMM3901208", nombre = "Taladro", precio =
699.00M,sucursal="Ferreteria S.A." };
Producto p6 = new Producto() { serieProducto = "SQAN1270008", nombre = "Escalera", precio =
959.99M,sucursal="Ferreteria S.A." };
//Lista de Productos
List<Producto> productos= new List<Producto>(){p1,p2,p3,p3,p4,p5,p6};

Cliente c1 = new Cliente() { idCliente = 1, nombre = "Brenda Liliana", edad = 22, direccion
= "Direccion cliente 1"};
Cliente c2 = new Cliente() { idCliente = 2, nombre = "Raul Hernandez", edad = 29, direccion
= "Direccion cliente 2"};
Cliente c3 = new Cliente() { idCliente = 3, nombre = "Karla Perez", edad = 32, direccion =
"Direccion cliente 3"};
Cliente c4 = new Cliente() { idCliente = 4, nombre = "Miguel Prado", edad = 51, direccion =
"Direccion cliente 4"};
Cliente c5 = new Cliente() { idCliente = 5, nombre = "Fernanda Itzel", edad =26, direccion
= "Direccion cliente 5"};
Cliente c6 = new Cliente() { idCliente = 6, nombre = "Mario Wargner", edad = 41, direccion
```

```
= "Direccion cliente 6"};
//Lista de Clientes
List<Cliente> clientes= new List<Cliente>(){c1,c2,c3,c4,c5,c6};

Orden orden1 = new Orden() { idOrden = 1, idCliente = c1.idCliente,
serieProducto=p1.serieProducto};
Orden orden2 = new Orden() { idOrden = 2, idCliente = c1.idCliente,
serieProducto=p2.serieProducto};
Orden orden3 = new Orden() { idOrden = 3, idCliente = c2.idCliente,
serieProducto=p3.serieProducto};
Orden orden4 = new Orden() { idOrden = 4, idCliente = c3.idCliente,
serieProducto=p4.serieProducto};
Orden orden5 = new Orden() { idOrden = 5, idCliente = c3.idCliente,
serieProducto=p5.serieProducto};
Orden orden6 = new Orden() { idOrden = 6, idCliente = c3.idCliente,
serieProducto=p6.serieProducto};
Orden orden7 = new Orden() { idOrden = 7, idCliente = c4.idCliente,
serieProducto=p2.serieProducto};
Orden orden8 = new Orden() { idOrden = 8, idCliente = c5.idCliente,
serieProducto=p1.serieProducto};
Orden orden9 = new Orden() { idOrden = 9, idCliente = c6.idCliente,
serieProducto=p6.serieProducto};
Orden orden10 = new Orden() { idOrden = 10, idCliente = c6.idCliente,
serieProducto=p1.serieProducto};
//Lista de Ordenes
List<Orden> ordenes = new List<Orden> (){ orden1, orden2, orden3, orden4, orden5,
orden6,orden7,orden8,orden9,orden10};
```

2.4.1 OPERACIONES DE PROYECCIÓN.

En una proyección se convierte un tipo de objeto a otro tipo y a veces, solo se toman aquellas propiedades que se utilizaran o que se quieren extraer de una consulta. Cuando se realiza una proyección, se puede crear un nuevo tipo a partir de cada objeto además de que se puede proyectar solo una propiedad y realizar una función sobre esta, aunque es válido proyectar el objeto original sin cambiarlo. Los métodos de operador de consulta estándar que realizan proyección se muestran en la siguiente tabla.

Nombre del método	Descripción	Sintaxis de las expresiones de consulta de C#
Select	Proyecta valores basados en una función de transformación.	select
SelectMany	Proyecta secuencias de valores basados en una función de transformación y los agrupa en una sola secuencia.	Utilizar varias cláusulas from

2.4.1.1 Select.

El ejemplo siguiente utiliza la cláusula *select* en *C#* para proyectar la primera letra de cada cadena de una lista de cadenas.

```
string[] palabras = new string[] { "Palabra", "Linq", "Astro", "En" };

IEnumerable<string> query = from palabra in palabras
                            select palabra.Substring(0, 1);

foreach (string s in query)
          Console.Write(s);

/* La salida producida es: PLAE */
```

Ejemplo 2:

```
IEnumerable<Orden> query = from q in ordenes
                           select q;

foreach(Orden item in query)Debug.WriteLine(" Id Orden:"+
```

```
item.idOrden+" Id Cliente:");
/* La salida es:
Id Orden:1 Id Cliente: Id Orden:2 Id Cliente: Id Orden:3 Id Cliente:
Id Orden:4 Id Cliente: Id Orden:5 Id Cliente: Id Orden:6 Id Cliente:
Id Orden:7 Id Cliente: Id Orden:8 Id Cliente: Id Orden:9 Id Cliente:
Id Orden:10 Id Cliente:* /
```

2.4.1.2 SelectMany.

El ejemplo siguiente utiliza varias cláusulas *from* en *C#* para proyectar cada letra de cada cadena contenidas en una lista de cadenas.

```
string[] palabras = new string[] { "Palabra", "Linq", "Astro", "En" };

IEnumerable<char> query = from palabra in palabras
                          from letra in palabra
                          select letra;
//O también
IEnumerable<char> query = palabras.SelectMany(x => x);

foreach (char letra in query)
    Console.WriteLine(letra);

/* La salida es:
P a l a b r a L i n q A s t r o E n */
```

Diferencias entre Select y SelectMany.

La forma de trabajar de *Select()* y *SelectMany()* se basa en crear un valor de resultado (o valores) a partir de los valores de origen. Por un lado, *Select()* genera un valor de resultado para cada valor de origen, es decir, que el numero de resultados obtenidos es el mismo que el numero de datos del origen, por lo que resultado total es una colección que tiene el mismo número de elementos que la colección de origen. Sin embargo, *SelectMany()* genera un resultado total único que contiene sub-colecciones concatenadas procedentes de cada valor de origen. La función que se pasa como un argumento a *SelectMany()* debe devolver una secuencia enumerable de valores para cada valor de origen. Estas secuencias enumerables se concatenan entonces mediante *SelectMany()* para crear una sola secuencia mayor.

2.4.2 OPERACIONES DE FILTRADO DE DATOS.

En el proceso de filtrado se limitan resultados devueltos con el fin de obtener solo los elementos que satisfacen una condición especificada por el programador. Los métodos de operador de consulta estándar que realizan la selección se muestran en la siguiente tabla.

Nombre del método	Descripción	Sintaxis de las expresiones de consulta de C#
OfType	Selecciona valores con el criterio de si pueden o no convertirse a un tipo especificado.	No es aplicable
Where	Selecciona valores basados en una función de condición.	where

2.4.2.1 OfType.

El uso de este operador se debe a que *LINQ* trabaja con colecciones que implementan la interfaz *IEnumerable<T>,* esto es por que se encuentran fuertemente tipadas. En el siguiente ejemplo se muestra como de un arreglo del tipo *object* se pueden seleccionar elementos de acuerdo a un tipo especificado:

```
object[] objetos = {"cadena1", 28, 12/12/2006, 3.3f, 1, "cadena2", clientes};
var query = objetos.OfType<string>();
foreach (string cadena in query) Console.WriteLine(cadena); }
/* salida:
   cadena1
   cadena2 */
```

Por ejemplo, en un *ArrayList* se puede tener tanto elementos de tipo *string* como de tipo *integer* o de otros tipos:

```
//Donde orden1=orden, p=producto y c= cliente
ArrayList lista = new ArrayList() { orden1, orden2, p1, p2, c1, c2 };
```

Con este método, se es capaz de convertir una colección que no implemente la interfaz *IEnumerable<T>*, en una que si lo haga. De esta manera es posible utilizar este tipo de colecciones dentro de LINQ, como en el siguiente ejemplo:

```
IEnumerable<Cliente> query = lista.OfType<Cliente>();

foreach (var item in query) Debug.WriteLine(item.nombre);
//El resultado es:
//Brenda Liliana c1
//Raul Hernandez c2
```

La diferencia que existe entre los dos métodos es que en el caso de *Cast*, si dentro de la colección se encuentra un elemento que no pueda ser convertido al tipo marcado, se lanzará una excepción. En el caso de *OfType*, si se encuentra un elemento que no se pueda convertir al tipo indicado, este se desecha.

2.4.2.2 Where.

En el ejemplo siguiente se utiliza la cláusula *where* en *C#* para filtrar de un arreglo las cadenas que tienen una longitud en específico.

```
string[] coleccion = { "uno", "dos", "tres", "cuatro", "aro" };

IEnumerable<string> query = from palabra in coleccion
                            where palabra.Length == 3
                            select palabra;

foreach (string cadena in query)
      Debug.WriteLine(cadena);
/* La salida producida es:
uno dos aro
*/
```

2.4.3 OPERACIONES DE ORDENACIÓN.

En esta operación los elementos de una secuencia son ordenados de acuerdo a uno o varias propiedades. El primer criterio de ordenamiento realiza una ordenación principal de los elementos, cuando se coloca un segundo criterio de ordenación, los elementos dentro de cada grupo ya ordenado, se vuelven a ordenar. Los métodos de operador de consulta estándar que ordenan los datos se muestran en la siguiente tabla.

Nombre del método	Descripción	Sintaxis de las expresiones de consulta de C#
OrderBy	Ordena una secuencia en función de los valores en orden ascendente	orderby
OrderByDescending	Ordena una secuencia en función de los valores en orden descendente	orderby … descending
ThenBy	Ordena una secuencia que ya se ha ordenado en orden ascendente	orderby …, …
ThenByDescending	Ordena una secuencia que ya se ha ordenado en orden descendente	orderby …, … descending
Reverse	Invierte el orden de los elementos de una colección	No es aplicable

2.4.3.1 OrderBy ascendente.

En el ejemplo siguiente se muestra cómo utilizar la cláusula *orderby* en una consulta *LINQ* para ordenar las cadenas de un arreglo por longitud de cadena, en orden ascendente.

```
// Crear la fuente de datos IEnumerable.
string[] palabras = { "linq", "operadores", "orderby", "arreglo", "palabra" };

//Se ha declarado una expresión de consulta en LINQ, no se ha usado la palabra var,
//que se sabe que el valor devuelto sera una colección que implementa a IEnumerable
IEnumerable<string> query = from palabra in palabras
                            orderby palabras.Length
                            select palabra;

foreach (string cadena in query)
    Console.WriteLine(cadena);

/* La salida que se produce es la siguiente:
linq operadores orderby arreglo palabra */
```

2.4.3.2 OrderByDescending.

En el ejemplo siguiente se muestra cómo utilizar la cláusula *orderby … descending* en una consulta *LINQ* para ordenar las cadenas por su primera letra, en orden descendente.

```
// Crear la fuente de datos IEnumerable.
string[]palabras = { "linq", "operadores", "orderby", "arreglo", "palabra"  };

// Crear la consulta. Aqui se  usó sintaxis de expresión de consulta
IEnumerable<string> query = from palabra in palabras
                            orderby palabra.Substring(0, 1) descending
                            select palabra;
//haciendo uso del mètodo como mètodo estàtico que forma parte de la fuente que
//implementa IEnumerable<T>en este caso un arreglo de cadenas
IEnumerable<string> query = palabras.OrderByDescending(x => x.Substring(0, 1));
foreach (string str in query)
    Console.WriteLine(str);

/* El resultado para ambas formas es la misma:
palabra operadores orderby linq arreglo */
```

2.4.3.3 Ordenación secundaria ascendente (ThenBy).

En el ejemplo siguiente se muestra cómo utilizar la cláusula *orderby* en una consulta *LINQ* para realizar una ordenación principal y secundaria de las cadenas de un arreglo. Las cadenas se ordenan primero por longitud (ordenación principal) y después por la primera letra de la cadena (ordenación secundaria), en ambos casos de forma ascendente.

```
// Crear la fuente de datos IEnumerable.
string[]palabras = { "linq", "operadores", "orderby", "arreglo", "palabra" };

// Crear la consulta. Aqui se  usó sintaxis de expresiónde consulta
IEnumerable<string> query = from palabra in palabras
                            orderby palabra.Length, palabra.Substring(0, 1)
                            select palabra;
//haciendo uso del mètodo de forma directa atravès de la clase Enumerable
//el primer parámetro de ThenBy es la fuente de tipo IOrderedEnumerable<T>, esta es producida
//al aplicar el método OrderBy (una secuencia de elementos ordenados), el segundo parámetro
//es una función selectora para aplicar a c/u de los elementos ordenados Fun<Tsource,TKey>
//en este caso Func<string,string>
IEnumerable<string> query = Enumerable.ThenBy(palabras.OrderBy(x => x.Length), x =>
x.Substring(0, 1));

foreach (string str in query)
    Console.WriteLine(str);
/* El resultado para ambas formas es la misma:
linq arreglo orderby palabra operadores */
```

2.4.3.4 Ordenación secundaria descendente (ThenByDescending).

En el ejemplo siguiente se muestra cómo utilizar la cláusula *orderby … descending* en una consulta *LINQ* para realizar una ordenación principal, en orden ascendente, y una ordenación secundaria, en orden descendente. Las cadenas se ordenan primero por longitud y después por la primera letra de la cadena.

```
// Crear la fuente de datos IEnumerable.
string[]palabras = { "linq", "operadores", "orderby", "arreglo", "palabra" };

// Crear la consulta.
IEnumerable<string> query = from palabra in palabras
                            orderby palabra.Length, palabra.Substring(0, 1) descending
                            select palabra;

//haciendo uso del mètodo de forma directa atravès de la clase Enumerable
//el primer parámetro de ThenByDescending es la fuente de tipo IOrderedEnumerable<T>
//producida al aplicar el método OrderBy el segundo parámetro es una función
//selectora para aplicar a c/u de los elementos ordenados Fun<Tsource,TKey>
//en este caso Func<string,string>
IEnumerable<string> query2 = Enumerable.ThenByDescending(palabras.OrderBy(x => x.Length),
                                    q=> q.Substring(0, 1));
foreach (string str in query)
    Console.WriteLine(str);

/* El resultado para ambas formas es la misma:
linq  palabra orderby arreglo operadores */
```

2.4.4 Operaciones de agrupación de datos.

Esta operación coloca los datos organizados en grupos para que los elementos de cada grupo compartan una propiedad en común. Los métodos de operador de consulta estándar que agrupan los elementos de datos se enumeran en la siguiente tabla.

Nombre del método	Descripción	Sintaxis de las expresiones de consulta de C#
GroupBy	Agrupa los elementos que comparten un atributo común. Cada grupo se representa mediante un objeto IGrouping(TKey, TElement).	group … by O bien, group … by … into …
ToLookup	Inserta elementos en un objeto Lookup(TKey, TElement) (diccionario uno a varios) según una función del selector de claves.	No es aplicable

2.4.4.1 GroupBy.

En el siguiente fragmento código se hace uso de la cláusula *group by* en *C#* para agrupar las palabras de una lista en función de si su longitud es mayor a 6.

```
string[] lista = new string[]{ "Alfredo", "Milla Jovovich", "Brenda", "Jose", "Tony",
                               "Maria Rosales" };

var query = from palabra in lista group palabra by palabra.Length > 6;

foreach (var item in query){
   Console.WriteLine(item.Key ? "Mayores: " : "Menores: ");
      foreach (var subitem in item){
         Console.WriteLine(subitem);
       }
}
/* La salida producida es:
Mayores: Alfredo, Milla Jovovich, Maria Rosales

Menores: Brenda, Jose, Tony */
```

2.4.5 OPERACIONES DE COMBINACIÓN (JOIN).

En esta operación, dos fuentes de datos son asociados a través de un atributo común que ambos comparten. Este tipo de operación es uno de los elementos importantes en las consultas de base de datos cuyas relaciones mutuas no se pueden realizar directamente. Estos métodos realizan combinaciones de igualdad, es decir, combinaciones que buscan las coincidencias entre dos orígenes de datos según la igualdad de sus atributos, en este caso de propiedades en común que tienen los dos orígenes de datos.
Los métodos de operador de consulta estándar que combinan los datos se muestran en la siguiente tabla.

Nombre del método	Descripción	Sintaxis de las expresiones de consulta de C#
Join	Combina dos secuencias según las funciones del selector de claves y extrae pares de valores.	join … in … on … equals …
GroupJoin	Combina dos secuencias según las funciones del selector de claves y agrupa las coincidencias resultantes para cada elemento.	join … in … on … equals … into …

En las bases de datos, un *join* (*inner join*) genera un conjunto de resultados en el que cada elemento de la primera colección aparece una vez por cada elemento al que corresponde de la segunda colección. Si un elemento de la primera colección no tiene ningún elemento correspondiente a la segunda colección, este no se agrega al conjunto de resultados.

2.4.5.1 Ejemplo de combinación simple.

En el siguiente ejemplo se tienen dos colecciones que contienen objetos de dos tipos definidos por el usuario, *Orden* y *Cliente*. La consulta utiliza la cláusula *join* en *C#* para buscar la concordancia entre objetos *Orden* y objetos *Cliente* a través de la propiedad *idCliente* para ambas clases. La cláusula *select* en *C#* define cómo son regresados los objetos (tipo). En este ejemplo, los objetos resultantes son tipos anónimos que consisten en el id de la orden, el nombre del producto, el nombre del cliente y la dirección del cliente.

```
// Recordando las listas declaradas al principio
var query = from orden in ordenes
            join cliente in clientes
            on orden.idCliente equals cliente.idCliente
           join producto in productos on orden.serieProducto
                                      equals producto.serieProducto
            select new {idOrden=orden.idOrden,
                        producto=producto.nombre,
                        cliente=cliente.nombre,
                        direccion=cliente.direccion};

foreach(var orden in query)
{
       Console.WriteLine(orden.idOrden+" , "+orden.producto+" , "+
                         orden.cliente+" , "+orden.direccion);
}
/* La salida producida es:
1 , Laptop , Brenda Liliana , Direccion cliente 1
2 , Camara , Brenda Liliana , Direccion cliente 1
3 , Disco DVD , Raul Hernandez , Direccion cliente 2
3 , Disco DVD , Raul Hernandez , Direccion cliente 2
4 , Lampara , Karla Perez , Direccion cliente 3
5 , Taladro , Karla Perez , Direccion cliente 3
6 , Escalera , Karla Perez , Direccion cliente 3
7 , Camara , Miguel Prado , Direccion cliente 4
8 , Laptop , Fernanda Itzel , Direccion cliente 5
9 , Escalera , Mario Wargner , Direccion cliente 6
10 , Laptop , Mario Wargner , Direccion cliente 6 */
```

2.4.6 OPERACIONES DE CUANTIFICACIÓN.

Estas operaciones devuelven un valor booleano el cual indica si algunos o todos los elementos de una secuencia satisfacen una condición. Los métodos de operador de consulta estándar que pertenecen a este grupo se muestran en la siguiente tabla.

Nombre del método	Descripción	Sintaxis de las expresiones de consulta de C#
All	Determina si todos los elementos de una secuencia satisfacen una condición.	No es aplicable
Any	Determina si alguno de los elementos de una secuencia satisface una condición.	No es aplicable
Contains	Determina si una secuencia contiene un elemento especificado.	No es aplicable

2.4.6.1 All.

En el siguiente ejemplo se utiliza el método de extensión *All* para devolver de una colección los productos cuyos precios sean menores a una cantidad especificada.

```
Producto [] productos= new Producto [] {
         new Producto {serie = "ZWCE0000908", nombre = "Camara", precio = 999.99M },
         new Producto {serie = "ZXXWN000908", nombre = "Lampara", precio = 399.99M },
         new Producto {serie = "ZWCE0000908", nombre = "Monitor", precio = 2999.99M },
         new Producto {serie = "XMQW0000908", nombre = "Bocinas", precio = 999.99M }
          };

bool productosBaratos = productos.All(producto =>producto.precio<400);

Console.WriteLine("{0} los productos son baratos.", productosBaratos ? "Todos" : "No todos");
Console.ReadLine();
//El resultado es: No todos los productos son baratos.
```

2.4.6.2 Any.

En este caso *Any* tiene un funcionamiento similar, pero esta vez determinará si algunos de los elementos cumplen alguna condición. En el siguiente ejemplo se usará el último fragmento de código presentado en *All*:

```
Producto[] productos = new Producto[] {
new Producto{serieProducto = "ZWCE0000908", nombre = "Camara", precio = 999.99M },
new Producto{serieProducto = "ZXXWN000908", nombre = "Lampara", precio = 399.99M },
new Producto{serieProducto = "ZWCE0000908", nombre = "Monitor", precio = 2999.99M },
new Producto{serieProducto = "XMQW0000908", nombre = "Bocinas", precio = 999.99M }
};

bool productosBaratos = productos.Any(producto => producto.precio < 400);

Console.WriteLine("{0} de los productos son baratos.", productosBaratos ? "Algunos"
: "No todos");
Console.ReadLine();
/*La salida es:
Algunos de los productos son baratos.
*/
```

2.4.7 OPERACIONES DE PARTICIÓN DE DATOS.

En este tipo de operación, una secuencia de entrada se divide en dos partes, sin reorganizar los elementos para posteriormente regresar una de las secciones. Los métodos de

operador de consulta estándar que realizan particiones de las secuencias se muestran siguiente tabla.

Nombre de método	Descripción	Sintaxis de las expresiones de consulta de C#
Saltar	Omite los elementos hasta una determinada posición de una secuencia.	No es aplicable
SkipWhile	Omite los elementos de una secuencia siempre que el valor de una condición especificada sea true y, a continuación, devuelve los elementos restantes.	No es aplicable
Take	Admite los elementos hasta una determinada posición de una secuencia.	No es aplicable
TakeWhile	Admite los elementos según una función de predicado hasta que un elemento no satisface la condición.	No es aplicable

2.4.7.1 Skip.

En el ejemplo del siguiente código se utiliza el método de extensión *Skip* en *C#* para omitir los cuatro primeros clientes de una lista de ordenes antes de devolver los clientes restantes de la lista.

```
List<Orden> ordenes = new List<Orden> { orden1, orden2, orden3, orden4, orden5, orden6 };

IEnumerable<Cliente> query = from q in ordenes.Skip(4) select q.cliente;
//O también
IEnumerable<Cliente> query = ordenes.Skip(4).Select(x=>x.cliente);

foreach (Cliente item in query) Console.WriteLine("id: "+item.id+" , nombre: "+item.nombre);
/*El resultado es:
id: 5 , nombre: Fernanda Itzel
id: 6 , nombre: Mario Wargner*/
```

2.4.7.2 Take.

En el ejemplo del siguiente código se utiliza el método de extensión *Take* tanto en una expresión de consulta en *C#* como un método de extensión, para devolver a los tres primeros clientes de una lista de órdenes.

```
List<Orden> ordenes = new List<Orden> { orden1, orden2, orden3, orden4, orden5, orden6 };

IEnumerable<Cliente> query = from q in ordenes.Take(3) select q.cliente;

Enumerable<Cliente> query = ordenes.Take(3).Select(q => q.cliente);

foreach (Cliente item in query)
Console.WriteLine("id: " + item.id + " , nombre: " + item.nombre);

/* La salida es:
id: 1 , nombre: Brenda Liliana
id: 2 , nombre: Raul Hernandez
id: 3 , nombre: Karla Perez */
```

2.4.7.3 TakeWhile.

En el ejemplo del siguiente código se utiliza el método de extensión *TakeWhile* tanto en una expresión de consulta como en una sintaxis de llamadas a métodos en *C#*, para devolver a los clientes de una lista de ordenes mientras el id del cliente sea menor a 3.

```
List<Orden> ordenes = new List<Orden> { orden1, orden2, orden3, orden4, orden5, orden6 };

IEnumerable<Cliente> query = from q in ordenes.TakeWhile(q => q.cliente.id < 3)
                             select q.cliente;
```

```
IEnumerable<Cliente> query = ordenes.TakeWhile(q => q.cliente.id < 3).Select(q =>
q.cliente);

foreach (Cliente item in query)
    Console.WriteLine("id: " + item.id + " , nombre: " + item.nombre);
/* La salida es:
id: 1 , nombre: Brenda Liliana
id: 2 , nombre: Raul Hernandez */
```

2.4.8 OPERACIONES DE CONJUNTOS.

En este tipo de operaciones, se crea un conjunto de resultados basado en si existen o no determinados elementos dentro de una u otras colecciones de objetos. Los métodos que pertenecen a este grupo y que realizan operaciones de conjuntos se muestran en la siguiente tabla.

Nombre del método	Descripción	Sintaxis de las expresiones de consulta de C#
Distinct	Quita los valores duplicados de una colección.	No es aplicable
Except	Devuelve la diferencia de conjuntos, es decir, los elementos de una colección que no aparecen en una segunda colección.	No es aplicable
Intersect	Devuelve la intersección de conjuntos, es decir, los elementos que aparecen en cada una de dos colecciones.	No es aplicable
Unión	Devuelve la unión de conjuntos, es decir, los elementos únicos que aparecen en cualquiera de las dos colecciones.	No es aplicable

2.4.8.1 Distinct.

En el siguiente ejemplo se utiliza el método *Distinct* en *C#* en una consulta *LINQ* para devolver los números únicos de una lista de enteros.

```
// Crear las fuentes de datos IEnumerable.
int[] numeros = new int[] { 1, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7,
                        8, 9, 10, 3, 44, 45, 67, 67, 67 };
// Crear la consulta. Se ha declarado una expresión de consulta junto con
//la llamada al método extensor Distinct, dado que no tiene una equivalencia
//en una expresión de consulta pudo haber sido de otra forma
IEnumerable<int> query = (from numero in numeros
                          select numeros).Distinct();

string numerosDistinct = null;
foreach (int cadena in query)
    numerosDistinct += " " + cadena.ToString();

Console.WriteLine(numerosDistinct);
//La salida producida es la siguiente:

//1 2 3 4 5 6 7 8 9 10 44 45 67
```

Lo mismo sucede con cadenas:

```
// Crear las fuentes de datos IEnumerable.
string[] cadenas = new string[]{"cadena","cadena","valor","valor",
                      "entero","booleano","booleano","booleano"};

IEnumerable<string> query = (from cadena in cadenas select cadena).Distinct();

foreach (string cadena in query)
    Console.WriteLine(cadena);

//La salida producida es la siguiente:
// cadena   valor   entero   booleano
```

2.4.8.2 Except.

En este ejemplo se muestra cómo utilizar *LINQ* para comparar dos listas de enteros y mostrar los números que se encuentren en numeros2 pero no numeros1 y viceversa.

```
// Crear las fuentes de datos IEnumerable.
int[] numeros1 = new int[] { 1, 2, 3, 4, 5, 6, 8 };
int[] numeros2 = new int[] { 1, 2, 3, 5, 6, 7, 8, 9, 10 };
// Crear la consulta. Aqui se  usó sintaxis de método
IEnumerable<int> exceptQuery = (from numero in numeros1
                                select numero).Except(numeros2);
// Ejecutar la consulta.
string numeros = null;
foreach (int cadena in query)
    numeros += " " + cadena.ToString();
Console.WriteLine(numeros);
/* Salida :   7 9 10     */

IEnumerable<int> exceptQuery2 = (from numero in numeros2
                                 select numero).Except(numeros1);
string numeros = null;
foreach (int cadena in exceptQuery2)
    numeros += " " + cadena.ToString();
Console.WriteLine(numeros);
/* Salida:   4 */
```

Lo mismo sucede con cadenas:

```
string [] cadenas1 = new string[]
          {"cadena","cadena","valor","valor","entero","booleano","booleano","booleano"};
string [] cadenas2 = new string[]
           {"linq","internet","musica","musica","doble","booleano","booleano","cadena"};

var query= (from q in cadenas1 select q).Except(cadenas2);
foreach(var item in query)System.Diagnostics.Debug.WriteLine(item);

//El resultado a mostrar es:

valor entero

var query= (from q in cadenas2 select q).Except(cadenas1);
foreach(var item in query)System.Diagnostics.Debug.WriteLine(item);
//El resultado a mostrar es: linq internet musica doble
```

2.4.8.3 Intersect.

La secuencia devuelta contiene los elementos que son comunes a las dos secuencias de entrada.

```
int[] numeros1 = new int[] { 1, 2, 3, 4, 5, 6, 8 };
int[] numeros2 = new int[] { 1, 2, 3, 5, 6, 7, 8, 9, 10 };
//Crear la consulta. Se usó una sintaxis de método
IEnumerable<int> exceptQuery = (from numero in numeros1
                                select numero).Intersect(numeros2);
string numeros = null;
foreach (int cadena in query)
    numeros += " " + cadena.ToString();
Console.WriteLine(numeros);
/* Salida:   1 2 3 5 6 8*/

IEnumerable<int> exceptQuery = (from numero in numeros2
                                select numero).Except(numeros1);
numeros = null;
foreach (int cadena in exceptQuery)
    numeros += " " + cadena.ToString();
Console.WriteLine(numeros);
/* Salida:  7 9 10 */
```

Lo mismo sucede con cadenas:

```
string [] cadenas1 = new string[]
          {"cadena","cadena","valor","valor","entero","booleano","booleano","booleano"};
string [] cadenas2 = new string[]
           {"linq","internet","musica","musica","doble","booleano","booleano","cadena"};

var query= (from q in cadenas1 select q).Intersect(cadenas2);
foreach(var item in query)System.Diagnostics.Debug.WriteLine(item);

//El resultado a mostrar es: cadena booleano
```

2.4.8.4 Unión.

En este ejemplo se combinan los elementos que contienen 2 listas de enteros y proyectar los resultados. Con esto se trata de mostrar cómo realizar una unión simple.

```
int[] numeros1 = new int[] { 1, 2, 3, 4, 5, 6, 8 };
int[] numeros2 = new int[] { 1, 2, 3, 5, 6, 7, 8, 9, 10 };

IEnumerable<int> exceptQuery = (from numero in numeros1
                                select numero).Union(numeros2);
string numeros = null;
foreach (int cadena in query)
    numeros += " " + cadena.ToString();
Console.WriteLine(numeros);
/* salida:   1 2 3 4 5 6 8 7 9 10 */
```

Lo mismo sucede con cadenas:

```
string [] cadenas1 = new string[]
          {"cadena","cadena","valor","valor","entero","booleano","booleano","booleano"};
string [] cadenas2 = new string[]
           {"linq","internet","musica","musica","doble","booleano","booleano","cadena"};

var query= (from q in cadenas1 select q).Union(cadenas2);
foreach(var item in query)System.Diagnostics.Debug.WriteLine(item);

//El resultado a mostrar serían todos los elementos contenidos en los
/2 arreglos de cadenas
```

2.4.9 OPERACIONES DE GENERACIÓN.

Para este tipo de operaciones se crea una nueva secuencia de valores. Los métodos de operador de consulta estándar que realizan la generación se muestran en la siguiente tabla.

Nombre del método	Descripción	Sintaxis de las expresiones de consulta de C#
DefaultIfEmpty	Reemplaza una colección vacía con una colección singleton con valores predeterminados.	No es aplicable
Empty	Devuelve una colección vacía.	No es aplicable
Range	Genera una colección que contiene una secuencia de números.	No es aplicable
Repeat	Genera una colección que contiene un valor repetido.	No es aplicable

2.4.9.1 DefaultIfEmpty.

En el siguiente ejemplo se crea una orden con valores por *default* (para cliente y producto) y se crea una lista vacía del tipo *Orden*, con la cuál se realiza una consulta para devolver los resultados por *default* si esta vacía:

```
//Creando un producto default
Producto pDefault = new Producto{
    serieProducto = "00000000000",
    nombre = "NA",
    precio = 00.00M,
    sucursal = "NA"
};
//Agregandolo a la lista productos
productos.Add(pDefault);

//Creando un cliente default
Cliente cDefault = new Cliente{
    idCliente = 0,
    nombre = "NA",
    edad = 0,
    direccion = "NA"
};
//Agregandolo a la lista clientes
clientes.Add(cDefault);

//Creando una orden default
Orden ordenDefault = new Orden{
    idOrden = 0,
    idCliente = cDefault.idCliente,
    serieProducto = pDefault.serieProducto
};

//Generando una lista de ordenes vacía
List<Orden> ordenes = new List<Orden> { };
var query = (from q in ordenes
             join a in clientes
             on q.idCliente equals a.idCliente
             select q).DefaultIfEmpty(ordenDefault);

foreach (var item in query) Console.WriteLine("Id orden: " + item.idOrden +
            " Id Cliente: " + item.idCliente + " Serie Producto: " + item.serieProducto);

//El resultado es:
//Id orden: 0 Id Cliente: 0 Serie Producto: 00000000000
```

2.4.9.2 Empy.

El siguiente ejemplo muestra como reasignar a la lista productos un valor del mismo tipo de la lista (*Producto*) vacía

```
productos = Enumerable.Empty<Producto>().ToList();
//La lista productos ahora está vacia
```

2.4.9.3 Range.

En el siguiente ejemplo se genera una secuencia de números del 1 al 10 y posteriormente mostrar su cuadrado:

```
//haciendo uso de la clase Enumerable
IEnumerable<int> query = Enumerable.Range(1, 10).Select(x => x * x);
//haciendo uso de una expresión de consulta con un mètodo estàtico
IEnumerable<int> query = from q in Enumerable.Range(1, 10) select q * q;
foreach (int numero in query) Console.WriteLine(numero);
/* El resultado es:
1 4 9 16 25 36 49 64 81 100
*/
```

2.4.10 OPERACIONES DE IGUALDAD.

En esta operación, dos secuencias de valores son determinadas iguales si sus elementos correspondientes son iguales y tienen el mismo número de elementos. Los métodos de operador de consulta estándar que realizan operaciones de igualdad se muestran en la siguiente tabla.

Nombre del método	Descripción	Sintaxis de las expresiones de consulta de C#
SequenceEqual	Determina si dos secuencias son iguales mediante la comparación de los elementos par a par.	No es aplicable

En el siguiente ejemplo se crean dos arreglos de enteros con los mismos números y posteriormente se realiza la comparación a través de *SecuenceEqual* el valor devuelto determinara si las secuencias son iguales.

```
int[] numeros1 = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int[] numeros2 = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

bool iguales = numeros1.SequenceEqual(numeros2);
Console.WriteLine(iguales ? "Las secuencias son iguales" :
                            "Las secuencias no son iguales");
//El resultado es:
//Las secuencias son iguales
```

2.4.11 OPERACIONES DE ELEMENTOS.

Las operaciones de elementos devuelven un único elemento específico de una secuencia. Los métodos de operador de consulta estándar que realizan operaciones de elementos se muestran en la siguiente tabla.

Nombre del método	Descripción	Sintaxis de las expresiones de consulta de C#
ElementAt	Devuelve el elemento situado en un índice especificado de una colección.	No es aplicable
ElementAtOrDefault	Devuelve el elemento situado en un índice especificado en una colección o un valor predeterminado si el índice está fuera del intervalo.	No es aplicable
First	Devuelve el primer elemento de una colección o el primer elemento que satisface una condición.	No es aplicable
FirstOrDefault	Devuelve el primer elemento de una colección o el primer elemento que satisface una condición. Devuelve un valor predeterminado si no existe tal elemento.	No es aplicable
Last	Devuelve el último elemento de una colección o el último elemento que satisface una condición.	No es aplicable
LastOrDefault	Devuelve el último elemento de una colección o el último elemento que satisface una condición. Devuelve un valor predeterminado si no existe tal elemento.	No es aplicable
Single	Devuelve el único elemento de una colección o el único elemento que satisface una condición.	No es aplicable
SingleOrDefault	Devuelve el único elemento de una colección o el único elemento que satisface una condición. Devuelve un valor predeterminado si no existe tal elemento o si la colección no contiene exactamente un elemento.	No es aplicable

Para los siguientes ejemplos considere la declaración de la lista genérica de productos:

2.4.11.1 ElementAt.

En este ejemplo se devuelve un solo producto en una posición especificada:

```
Producto producto = (from prod in productos select prod).ElementAt(0);
Console.WriteLine("El producto es :" + producto.nombre);
//El resultado es:
// El producto es :Laptop
```

2.4.11.2 ElementAtOrDefault.

En el siguiente ejemplo se muestra cómo hacer uso del método *ElementAtOrDefault* en *C#* para determinar si en el índice especificado existe un producto, la realizar esto no se lanza ninguna excepción sobre un índice fuera del rango.

```
int indice = 45;
Producto producto = (from prod in productos select prod).ElementAtOrDefault(indice);
Console.WriteLine(producto == null ? "No existen elementos en el indice " + indice : "El
producto es :" + producto.nombre);
//El resultado es:
//No existen elementos en el indice 45
```

2.4.11.3 First.

En el siguiente ejemplo se devuelve solamente el primer elemento de la lista *productos*:

```
Producto producto = (from prod in productos select prod).First();
Console.WriteLine("El producto es :" + producto.nombre);
//El resultado es:
// El producto es :Laptop
```

2.4.11.4 FirstOrDefault.

En el siguiente ejemplo se emplea la misma lógica que en *ElementAtOrDefault*, pero esta vez aplicado a *FirsOrDefault*.

```
productos = Enumerable.Empty<Producto>().ToList();
Producto producto = (from prod in productos select prod).FirstOrDefault();
Console.WriteLine(producto == null ? "No existen elementos en la lista " : "El primer
producto es :" + producto.nombre);
//El resultado es:
//No existen elementos en la lista
```

2.4.11.5 Last.

En este ejemplo se hace uso de este método para devolver el último elemento de la lista *productos*:

```
Producto producto = (from prod in productos select prod).Last();
Console.WriteLine("El ultimo producto de la lista es: " + producto.nombre);
//El resultado es:
// El ultimo producto de lista es: Escalera
```

2.4.11.6 LastOrDefault.

Aquí se emplea la misma lógica para este tipo de método:

```
productos = Enumerable.Empty<Producto>().ToList();
Producto producto = (from prod in productos select prod).LastOrDefault();

Debug.WriteLine(producto == null ? "No existen elementos en la lista " : "El ultimo
producto de la lista es :" + producto.nombre);
//El resultado es:
//No existen elementos en la lista
```

2.4.11.7 Single.

En este ejemplo como en los anteriores se hace uso del método con el mismo nombre en una expresión de consulta en *C#* para devolver un único elemento de acuerdo a una condición:

```
Producto producto = (from prod in productos select prod)
                    .Single(x => x.serieProducto == "WXMM3901208");
Console .WriteLine("El producto es : " + producto.nombre);
//El resultado es:
//El producto es : Taladro
```

2.4.12 OPERACIONES DE AGREGACIÓN.

En esta operación se calcula un solo valor que representa un promedio, a partir de una colección de valores. Un ejemplo de este tipo de operación sería calcular la temperatura diaria promedio a partir de los valores de temperatura diarios de un mes completo.

Los métodos de operador de consulta estándar que realizan operaciones de agregación se muestran en la siguiente tabla.

Nombre del método	Descripción	Sintaxis de las expresiones de consulta de C#
Aggregate	Realiza una operación de agregación personalizada con los valores de una colección.	No es aplicable
Average	Calcula el valor promedio de una colección de valores.	No es aplicable
Count	Cuenta los elementos de una colección y, opcionalmente, sólo aquéllos que satisfacen una función de predicado.	No es aplicable
LongCount	Cuenta los elementos de una colección grande y, opcionalmente, sólo aquéllos que satisfacen una función de predicado.	No es aplicable
Max	Determina el valor máximo de una colección.	No es aplicable
Min	Determina el valor mínimo de una colección.	No es aplicable
Sum	Calcula la suma de los valores de una colección.	No es aplicable

2.4.12.1 Average.

En el siguiente código se utiliza el método *Average* en *C#* para calcular la temperatura promedio en un arreglo de números.

```
double[] temperaturas = { 72.0, 81.5, 69.3, 88.6, 80.0, 68.5 };

double promedio = temperaturas.Average();
Console.WriteLine(promedio);
//El resultado es:
//76.65
```

En el siguiente ejemplo se calcula el promedio en el precio de la lista que contiene ciertos productos:

```
decimal promedioProductos = (from q in productos select q.precio).Average();
Console.WriteLine(promedioProductos);
//El resultado es:
//508.13
```

2.4.12.2 Count.

En el ejemplo de código siguiente se utiliza el método *Count* en *C#* para contar el número de valores de una lista de productos que son mayores que 300.

```
int numeroProductos = (from prod in productos select prod).Count(x => x.precio > 300);
//ó tambien
int numeroProductos = productos.Count(x => x.precio > 300);

Debug.WriteLine(numeroProductos);
//La salida es:
// 4
```

2.4.12.3 Max.

En el ejemplo de código siguiente se utiliza el método *Max* en *C#* para calcular el precio máximo en una lista de productos.

```
decimal valorMaximo = (from prod in productos select prod.precio).Max();
//O tambien
decimal valorMaximo = productos.Max(x => x.precio);

Console.WriteLine(valorMaximo);
//La salida es:
// 999.99
```

2.4.12.4 Sum.

En el ejemplo de código siguiente se utiliza el método de extensión Sum para calcular el total de los precios de todos los productos.

```
decimal sumaPrecios = (from prod in productos select prod.precio).Sum();
//O tambien
decimal sumaPrecios = productos.Sum(x => x.precio);

Console.WriteLine(sumaPrecios);
//La salida es:
// 3556.9
```

2.4.13 OPERACIONES DE CONVERSIÓN DE TIPOS DE DATOS.

En este tipo de operaciones el tipo de los objetos de una secuencia de entrada es cambiado a otro tipo especificado. Este tipo de operaciones pueden resultar útiles en las consultas en diferentes contextos, como por ejemplo:

- El método *Enumerable.AsEnumerable(TSource)* puede usarse para no mostrar la implementación personalizada de un tipo de un operador de consulta estándar.
- El método *Enumerable.OfTypeTResult)* se puede utilizar para habilitar las colecciones no parametrizadas para las consultas LINQ.
- Los métodos *ToArray(TSource)*, *ToDictionary*, *ToList(TSource)* y *ToLookup* de *Enumerable* se pueden utilizar para forzar la ejecución inmediata de la consulta, en lugar que sea ejecutada cuando sean iterados sus elementos.

En la siguiente tabla se muestran los métodos que realizan este tipo de operaciones. Los métodos de conversión que se muestran y cuyo nombre empieza por "*As*" cambian el tipo estático de la colección de origen pero no iteran sobre ella, es decir, no realizan una ejecución inmediata. Los métodos cuyos nombres empiezan por "*To*" iteran sobre la colección de origen y colocan los elementos en el tipo de colección correspondiente.

Nombre del método	Descripción	Sintaxis de las expresiones de consulta de C#
AsEnumerable	Devuelve la entrada tipificada como IEnumerable<T>.	No es aplicable
AsQueryable	Convierte una interfaz IEnumerable (genérica) en una interfaz IQueryable (genérica).	No es aplicable
Conversión de tipos explícita	Convierte los elementos de una colección a un tipo especificado.	Utilice una variable de rango con tipo explícito. Por ejemplo: from string str in words
ToArray	Convierte una colección en una matriz. Este método fuerza la ejecución de la consulta.	No es aplicable
ToDictionary	Coloca los elementos en Dictionary(TKey, TValue) según una función del selector de claves. Este método fuerza la ejecución de la consulta.	No es aplicable

ToList	Convierte una colección a List<T>. Este método fuerza la ejecución de la consulta.	No es aplicable
ToLookup	Coloca los elementos en un objeto Lookup (TKey, TElement) (diccionario uno a varios) según una función del selector de claves. Este método fuerza la ejecución de la consulta.	No es aplicable

En el ejemplo de código siguiente se utiliza una variable de rango con tipo explícito en *C#* para convertir un tipo a un subtipo antes de tener acceso a un miembro que sólo está disponible en el subtipo.

```
class Planta
{ public string Nombre { get; set; } }

class PlantaCarnivora : Planta
{ public string Tipo { get; set; } }

class Linq
{
    static void Main(string[] args)
    {
        Planta[] plantas = new Planta[] {
    new PlantaCarnivora { Nombre = "Planta 1", Tipo = "Tipo 1" },
    new PlantaCarnivora { Nombre = "Planta 2", Tipo = "Tipo 2" },
    new PlantaCarnivora { Nombre = "Planta 3", Tipo = "Tipo 3" },
    new PlantaCarnivora { Nombre = "Planta 4", Tipo = "Tipo 4" }
};

        var query = from PlantaCarnivora plantCar in plantas
                    //Sin esta conversión plantaCar
                    //no contiene una definición de Tipo:
                    where plantCar.Tipo == "Tipo 2"
                    select plantCar;

        foreach (Planta p in query) Console.WriteLine(p.Nombre);
    }
}
//El resultado es:
// Planta 2
```

2.4.13.1 *Cast*.

Para el siguiente ejemplo se emplean las mismas clases que en el anterior y la misma lógica de funcionamiento:

```
var query = from plantCar in plantas.Cast<PlantaCarnivora>()
            where plantCar.Tipo == "Tipo 2"
            select plantCar;

foreach (Planta p in query) Console.WriteLine(p.Nombre);
//El resultado es:
// Planta 2
```

2.4.14 OPERACIONES DE CONCATENACIÓN.

En este tipo de operación se realiza la concatenación de una secuencia con otra. Los métodos de operador de consulta estándar que realizan la concatenación se muestran en la siguiente tabla.

Nombre del método	Descripción	Sintaxis de las expresiones de consulta de C#
Concat	Concatena dos secuencias para formar una secuencia.	No es aplicable

2.5 SERIALIZACIÓN.

La serialización es el proceso mediante el cuál se convierte el estado de un objeto para que se pueda almacenar o transportar. El caso contrario de este proceso es la deserialización, que transforma una secuencia a un objeto, ambos procesos permiten almacenar los datos y transferirlos con facilidad.
.NET Framework incorpora dos procesos de serialización:

- La serialización binaria. Este tipo conserva fidelidad de tipo para conservar el estado de un objeto entre las invocaciones diferentes de una aplicación. Por ejemplo, se puede compartir un objeto entre distintas aplicaciones si es serializada en el Portapapeles. Se puede serializar un objeto en una secuencia, un disco, la memoria, en de la red, etc. La comunicación remota utiliza la serialización para pasar objetos "por valor" de un equipo o una aplicación a otros.
- La serialización *XML*. Este tipo serializa sólo propiedades públicas y campos y no conserva la fidelidad de tipo. Esto es para proporcionar o utilizar los datos sin restringir la aplicación que utiliza los datos. Dado que *XML* es un estándar abierto, es una opción interesante para compartir los datos por a través de la Web.

LINQ to SQL proporciona la habilidad de realizar una carga aplazada en forma predeterminada, este tipo de carga se refiere al proceso de recuperar únicamente el objeto solicitado cuando se realiza una consulta, por lo que no se capturan los objetos relacionados al mismo tiempo. No existe una manera se conocer si los objetos que se requieren se encuentran en memoria, ya que si se intenta acceder a estos, se crea una solicitud que los recupera automáticamente. Con *Windows Communication Foundation* (*WCF*) o cualquier método que traslade objetos de un proceso a otro, éstos deben ser serializados, en específico las clases resultantes de una capa de datos (clases, métodos, propiedades, etc) proveniente de una base de datos que nos permitan obtener todos los datos. La característica de serialización de *LINQ to SQL* resuelve este problema, principalmente a través de dos mecanismos:

- Un modo *DataContext* para desactivar la carga aplazada.
- Un modificador de generación de código para generar los atributos *System.Runtime.Serialization.DataContractAttribute* y *System.Runtime.Serialization.DataMemberAttribute* en las entidades generadas.

El código siguiente utiliza las clases *Cliente* y *Orden* de una la base de datos *BaseDeDatos* y muestra cómo en estas clases se agregan atributos de serialización.

```
[Table(Name = "dbo.Clientes")]

//Atributo DataContract
[DataContract()]

public partial class Cliente : INotifyPropertyChanging, INotifyPropertyChanged { }
```

En este caso *DataContract* es un serializador predeterminado utilizado por el componente *WCF* de *.NET Framework 3.0* o versiones posteriores.

2.6 EXTENSIONES DE LINQ.

2.6.1 LINQ TO XML.

El lenguaje *XML* es ampliamente usado para dar formato a datos en diversas formas, este lenguaje se encuentra ampliamente extendido, tanto que puede localizarse en la web, en archivos de configuración, en archivos de diferente proveedor y en bases de datos. *LINQ to XML* es una

interfaz de programación *XML* (espacio de nombres *System.Xml.Linq*) para *LINQ* que permite trabajar sobre *XML* desde los lenguajes de programación nativos de *.NET Framework* y con esto poder modificar un documento en memoria que puede guardarse en un archivo o serializarlo y enviarlo a través de Internet.

La integración de *LINQ to XML* con *LINQ* permite desarrollar consultas sobre documentos *XML* en memoria para posteriormente obtener como resultado colecciones de elementos y atributos. Además, estos resultados pueden usarse como parámetros en constructores de objetos como *XElement* y *XAttribute*, propios de *LINQ to XML*, con lo que se pueden crear árboles *XML* de una forma más eficaz, de esta forma se permite a los desarrolladores convertir árboles *XML* de una forma a otra.

2.6.2 LINQ TO ENTITIES.

En la actualidad, las aplicaciones que se desarrollan hacen uso de bases de datos, por lo que estas aplicaciones inevitablemente tienen que interactuar con los datos contenidos en estas bases de datos. Los esquemas usados por las bases de datos son siempre difíciles de abstraer al momento de la creación de las aplicaciones, y los modelos conceptuales de las aplicaciones difieren de los modelos lógicos de las bases de datos. El modelo de datos conceptual *Entity Data Model* (*EDM*) es un modelo que se puede usar para representar los datos relacionales para que las aplicaciones que se desarrollen puedan interactuar con los datos como entidades u objetos.

A través de este modelo, se usa *ADO.NET* para manipular los datos relacionales como objetos propios de *.NET*, esto hace que la capa de acceso a datos sea un objetivo ideal en el que se pueda hacer uso de *LINQ*. Para poder hacer uso de *LINQ To Entities*, primero debe hacerse una referencia a los ensamblados *System.Core.dll* y *System.Data.Entity.dll*, y a los espacios de nombres *System.Linq* y *System.Data.Objects*, además de estos requisitos, se requiere una referencia al modelo *Entity Data Model* (*EDM*) que se va a consultar.

Muchos de los operadores de consulta estándar de *LINQ* tienen una versión sobrecargada que acepta un argumento entero. El argumento entero corresponde a un índice que comienza en cero en la secuencia en la que se opera, un *IEqualityComparer* o *IComparer*. A menos que se especifique lo contrario, estas versiones sobrecargadas de los operadores de consulta estándar de *LINQ* no se admiten y, si se intenta utilizarlos, se iniciará una excepción.

2.6.3 LINQ TO SQL.

LINQ to SQL a diferencia de *LINQ To Entities*, no requiere de una asignación a un modelo conceptual, ya que se puede usar el modelo de programación de *LINQ* directamente en un esquema de base de datos existente. *LINQ to SQL* permite a los programadores generar clases de *.NET Framework* que representen a los datos, así, esas clases generadas se asignan directamente a tablas de bases de datos, vistas, procedimientos almacenados y funciones definidas por el usuario.

En *LINQ to SQL*, el modelo de datos de una base de datos se asigna a un modelo de objetos que corresponda al lenguaje de programación en *.NET*. Posteriormente, cuando se ejecuta la aplicación, *LINQ to SQL* convierte al lenguaje SQL las consultas realizadas en el lenguaje que correspondan al modelo de objetos y las envía a la base de datos para su ejecución. Una vez que las consultas hayan sido procesadas y la base de datos devuelve los resultados, el motor de *LINQ to SQL* los vuelve a convertir en objetos compatibles con .NET y con el propio lenguaje. Por ejemplo, en el siguiente código se crea el objeto *bd* para representar la base de datos *BaseDeDatos*, el destino es la tabla *Clientes*, las filas se filtran para *Clientes* de Mexico y se selecciona para la recuperación una cadena de *NombreCompania*. Cuando se ejecuta la consulta al iterar sobre ella con la sentencia *foreach*, se recupera la colección de valores *NombreCompania*.

```
BaseDeDatos BD = new BaseDeDatos(@"ruta\BaseDeDatos.mdf");

var query = from cliente in BD.Clientes
            where Cliente.Cliudad == "Mexico"
            select cliente.NombreCompania;

foreach (var cliente in query) {
    Console.WriteLine(cliente);
}
```

LINQ to SQL incorpora las operaciones necesarias que se requieren para manipular los datos como se hace en *SQL*. Se puede consultar, insertar, actualizar y eliminar información en las tablas, esto y más se observará en el siguiente capítulo (3). Al tener un modelo de objetos, se deben describir las peticiones de información y manipular los datos dentro de ese modelo, de esta forma, los datos se manejan en como objetos y propiedades que corresponden al modelo de objetos, y no como filas y columnas correspondientes a la base de datos, por lo que evidentemente no se trabaja de forma directa sobre la base de datos. Cuando se indica a *LINQ to SQL* que ejecute una consulta escrita o se llama al método *SubmitChanges()* sobre los datos que se manipulan, *LINQ to SQL* se comunica con la base de datos en el lenguaje de la misma. Esto solo es una visión general de lo que se puede hacer con esta extensión (*LINQ To SQL*), en el siguiente capítulo se tomará este y otros conceptos que se encuentran involucrados.

2.7 PROVEEDORES DE CONSULTA DE TERCEROS.

2.7.1 DEVART.

LINQ To SQL opera solamente con *SQL Server*. En este caso *Devart* es una empresa que está dedicada al desarrollo de software especializado, centrándose en dar soluciones de conectividad nativa y herramientas de desarrollo para las bases de datos más conocidas como lo son *Oracle*, *SQL Server*, *MySQL*, *PostgreSQ* y otros. Entre las diversas herramientas que desarrolla se encuentra *dotconnect for ...* seguido el nombre conocido del *SMBD* (*MySQL,Oracle* y *PostgreSQL*) la cuál implementa la misma idea de *LINQ To SQL*.

1. Devart: *LINQ To Oracle* (*dotconnect for Oracle*)
2. Devart: *LINQ To MySQL* (*dotconnect for mysql*)
3. Devart: *LINQ To PostgreSQL* (*dotconnect for PostgreSQL*)

Todas estas herramientas se pueden localizar y descargar en su dirección principal, aunque desde un inicio se distribuyen bajo la licencia Trial, así que si se requiere hacer un uso extendido de estas herramientas debe pagarse. Para tener soporte, se debe tener la versión 3.5 de *.NET FrameWork* (donde viene incluido *LINQ*) y *Visual Studio 2008*. Actualmente se tienen 2 versiones, la primera es la *express* la cual incluye elementos básicos tales como clases propias para manejar los diversos *SMBD* (tipos de datos, estructuras), soporte para *.NET Framework*, etc, en el caso de la versión profesional se incluyen otras herramientas como un diseñador de objetos relacionales para mapear las bases de datos (al igual que como sucede en *Visual Studio 2008*) que son específicas del *SMBD* hacia clases enteras del *.NET*, controles que simplifican la conexión a una base de datos: *MySqlConnection,MySqlCommand*, soporte para *ASP.NET 2.0*, etc. Para hacer uso de estas herramientas solo se necesita instalarlo, todas las herramientas se incrustarán en el *IDE* de *Visual Studio*, haciéndolo más fácil de usar ya que los conceptos de *LINQ To SQL* también aplican para estas como la inserción, actualización y eliminación de datos ya que estas herramientas hacen una implementación de *LINQ* con clases de *.NET*.

2.7.2 DB_LINQ.

Esta es una herramienta para el mapeo de objetos relacionales, lo que significa que puede mapear la estructura de una base de datos hacia clases que entienda el *CLR* de *.NET*. Lo cual lo hace compatible con *LINQ*. Para generar el archivo de código correspondiente en el lenguaje de *.NET* (p.ej. *C#*) de una base de datos debe de ejecutarse un *script* y posteriormente agregarse una referencia del ensamblado *DbLinq.Mysql.dll* (en el caso de que los datos con los que se requieran trabajar provengan de *MySQL*). La ventaja es que se distribuye bajo la licencia *MIT* (*Massachusetts Institute of Technology*) lo que le da la característica de *OpenSource* y permite su modificación, no obstante esto puede no ser recomendable, además de contar con soporte para otros *SMBD*.

CAPITULO III. CONCEPTOS BÁSICOS SOBRE EL MANEJO DE BASE DE DATOS RELACIONALES CON LINQ TO SQL.

3.1 MAPEO DE OBJETOS RELACIONALES A OBJETOS.

Como se mencionó en el capitulo anterior, un modelo de objetos en un lenguaje de programación es asignado a una base de datos relacional. Existen diferentes formas de generar automáticamente un modelo de *Visual Basic* o *C#* a partir de los datos de una base de datos existente:

- Asignación directa basada en atributos (a través de código).
- Herramienta de línea de comandos *SQLMetal*.
- Generar el modelo de objetos como un archivo externo.
- El *IDE Visual Studio 2008* incorpora un diseñador relacional de objetos para generar un modelo de objetos. Este diseñador proporciona una interfaz de usuario que ayuda a generar un modelo de objetos de forma automática en *LINQ to SQL*.

En la ejemplificación de los siguientes temas, se hará uso de la base de datos *BDEjemplo* previamente diseñada, la cuál a grandes rasgos tiene la siguiente definición:

Libro		
IdLibro	int	not null
Titulo	nvarchar(100)	not null
IdAutor	int	not null
IdEditorial	int	not null
FechaPublicación	datetime	not null
Ejemplares	int	null
Notas	nvarchar(200)	null
Primary Key	IdLibro	
Foreign Key	IdAutor,IdEditorial	

Editorial		
IdEditorial	int	not null
Nombre	nvarchar(100)	not null
SitioWeb	nvarchar(100)	null
Direccion	nvarchar(200)	not null
Primary Key	IdEditorial	
Foreign Key	---	

Autor		
IdAutor	int	not null
Nombre	nvarchar(20)	not null
Apellidos	nvarchar(30)	not null
Primary Key	IdAutor	
Foreign Key	---	

3.1.1 ASIGNACIÓN BASADA EN ATRIBUTOS.

En esta parte se describe la asignación basada en atributos en su forma más elemental, en la que *LINQ to SQL* asigna una base de datos a un objeto *DataContext*, una tabla a una clase y las columnas y relaciones a las propiedades de esas clases. Cuando se usa *Visual Studio*, normalmente se realiza esta asignación mediante el diseñador relacional de objetos o se puede incluir los atributos en el código de forma directa. Para ello basta con incluir el espacio de nombres *System.Data.Linq.Mapping*.

3.1.1.1 Atributos.

A través de los atributos se agrega información descriptiva (tipos, métodos, propiedades, serialización de datos, etc.) al código del programa escrito en cierto lenguaje de *.NET*, esta información puede contener instrucciones del compilador o descripciones propias de los datos, los cuáles afectan el comportamiento de la aplicación cuando es ejecutada. Cuando se asocia un atributo a un objeto perteneciente al programa, la información que corresponda a este se puede

consultar en tiempo de ejecución a través de una técnica denominada reflexión. La reflexión proporciona objetos (de tipo *Type*) que encapsulan ensamblados, módulos y tipos. A través de esta reflexión, se crea dinámicamente una instancia de un tipo, se relaciona el tipo a un objeto existente y se obtiene el tipo a partir del objeto, también es posible llamar a sus métodos o tener acceso a sus campos y propiedades. Existen dos formas de atributos: atributos que se encuentran en la *BCL* del *CLR* y atributos personalizados que se pueden crear para agregar información adicional al código. Esta información se puede recuperar después mediante programación. Un ejemplo sencillo es el que se muestra a continuación, por ejemplo, el atributo *System.Reflection.TypeAttributes.Serializable* se utiliza para aplicar la característica de serialización a una clase denominada *ClaseEjemplo*:

```
[System.Serializable]
public class ClaseEjemplo {
    // Los objetos de este tipo pueden ser serializados.
}
```

Entre los atributos más importantes y usados en *LINQ To Sql* se encuentran:

3.1.1.2 Atributo DatabaseAttribute.

Este atributo se utiliza para especificar el nombre predeterminado de la base de datos cuando en la conexión no se proporciona ningún nombre. Este atributo es opcional, pero, si se utiliza, se debe aplicar la propiedad *Name*, como se indica en la siguiente tabla:

Propiedad	Tipo	Default	Descripción
Name	string	Name.	Cuando se usa con su propiedad Name, se especifica el nombre de la base de datos.

En el siguiente fragmento de código se asocia este atributo a la clase *BDEjemploDataContext*, y se especifica el nombre de la base de datos a través de la propiedad *Name*:

```
[System.Data.Linq.Mapping.DatabaseAttribute(Name="BDEjemplo")]
public partial class BDEjemploDataContext : System.Data.Linq.DataContext
        {
```

3.1.1.3 Atributo TableAttribute.

Con este atributo se designa una clase como una clase de entidad que está asociada a una tabla o vista de base de datos. *LINQ to SQL* trata las clases que tienen este atributo como clases persistentes. En la siguiente tabla se describe la propiedad *Name*:

Propiedad	Tipo	Default	Descripción
Name	string	La misma cadena que el nombre de clase	Designa una clase como una clase de entidad que está asociada a una tabla de base de datos.

En el siguiente fragmento de código, se utiliza este atributo para asignar a la clase parcial *Libro* como una entidad, la cuál va a estar asociada a una tabla llamada *Libro* contenida en la base de datos a través de la propiedad *Name*:

```
[Table(Name="dbo.Libro")]
public partial class Libro : INotifyPropertyChanging, INotifyPropertyChanged
        {
```

3.1.1.4 Atributo ColumnAttribute.

Este atributo se utiliza para designar un miembro de una clase de entidad para que represente una columna de una tabla de la base de datos. Este atributo se puede aplicar a cualquier campo o propiedad. Sólo los miembros que sean declarados como columnas se

recuperarán y conservarán cuando *LINQ to SQL* guarde los cambios en la base de datos. Los miembros que no tienen este atributo no son persistentes y por lo tanto no se envían para operaciones de inserción o actualización.
En la siguiente tabla se describen las propiedades de este atributo.

Propiedad	Tipo	Default	Descripción
AutoSync	AutoSync	Nunca	Indica al CLR que recupere el valor después de una operación de inserción o actualización. Opciones: Always, Never, OnUpdate, OnInsert.
CanBeNull	Boolean	**true**	Indica que una columna puede contener valores nulos.
DbType	string	Tipo de columna de base de datos deducido	Utiliza tipos de base de datos y modificadores para especificar el tipo de la columna de base de datos.
Expression	string	Vacío	Define una columna calculada en una base de datos.
IsDbGenerated	Boolean	**false**	Indica que una columna contiene valores que la base de datos genera automáticamente.
IsDiscriminator	Boolean	**false**	Indica que la columna contiene un valor de discriminador para una jerarquía de herencia de LINQ to SQL.
IsPrimaryKey	Boolean	**false**	Especifica que este miembro de clase representa una columna que es o forma parte de las claves principales de la tabla.
IsVersion	Boolean	**false**	Identifica el tipo de columna del miembro como una marca de tiempo o número de versión de la base de datos.
UpdateCheck	UpdateCheck	**Always**, a menos que IsVersion sea **true** para un miembro	Especifica cómo se plantea LINQ to SQL la detección de conflictos de simultaneidad dinámica.

En el siguiente fragmento de código se muestran dos propiedades de la clase parcial Libro, los cuáles tienen asociados, cada uno, el atributo *Column* y dependiendo de la definición que tengan las columnas en la base de datos, las propiedades usadas del atributo *Column* (*DbType*,*IsPrimaryKey*,*CanBeNull*) tendrán un valor asignado correspondiente a esa definición:

```
private int _IDLibro;
private string _Titulo;

Column(Storage="_IDLibro", DbType="Int NOT NULL", IsPrimaryKey=true)]
public int IDLibro
{get; set;}

[Column(Storage="_Titulo", DbType="NVarChar(100) NOT NULL", CanBeNull=false)]
public string Titulo
{get; set;}
```

3.1.1.5 Atributo AssociationAttribute.

Con este atributo se designa una asociación en la base de datos a una propiedad dentro de la clase entidad como una relación entre clave externa y la clave principal. En la siguiente tabla se describen las propiedades de este atributo.

Propiedad	Tipo	Default	Descripción
DeleteOnNull	Boolean	**false**	Cuando se coloca en una asociación cuyos miembros de clave externa no aceptan valores Null, elimina el objeto cuando la asociación está establecida en null.

DeleteRule	string	Ninguna	Agrega comportamiento de eliminación a una asociación.
IsForeignKey	Boolean	**false**	Si es verdadero, designa el miembro como la clave externa de una asociación que representa una relación de base de datos.
IsUnique	Boolean	**false**	Si es verdadero, indica una restricción de unicidad en la clave externa.
OtherKey	string	Identificador de la clase relacionada.	Designa uno o más miembros de la clase de entidad de destino como valores de clave en el otro lado de la asociación.
ThisKey	string	Identificador de la clase contenedora	Designa miembros de esta clase de entidad para que representen los valores de clave en este lado de la asociación.

En el siguiente fragmento de código se utiliza este atributo hacia una propiedad denominada Autor para poder establecer una relación entre llaves primarias y foráneas. Las asignaciones que tienen las propiedades de este atributo para la propiedad, son los mismos que se tienen en la base de datos:

```
private EntityRef<Autor> _Autor;

[Association(Name="Autor Libro", Storage=" Autor", ThisKey="IDAutor", IsForeignKey=true,
DeleteOnNull=true, DeleteRule="CASCADE")]
public Autor Autor
{get{...}set{...}}
```

3.1.1.6 Atributo FunctionAttribute.

Este atributo se utiliza para designar a un método que representa a un procedimiento almacenado o una función definida por el usuario en la base de datos. En la siguiente tabla se describen las propiedades de este atributo.

Propiedad	Tipo	Default	Descripción
IsComposable	Boolean	**false**	Si es falso, indica la asignación a un procedimiento almacenado. Si es verdadero, indica la asignación a una función definida por el usuario.
Name	string	La misma cadena que el nombre en la base de datos	Especifica el nombre del procedimiento almacenado o la función definida por el usuario.

3.1.1.7 Atributo ParameterAttribute.

Este atributo se utiliza para asignar parámetros de entrada en métodos de procedimiento almacenado a métodos que tienen designado el atributo *FunctionAttribute*. En la siguiente tabla se describen las propiedades de este atributo.

Propiedad	Tipo	Default	Descripción
DbType	string	Ninguna	Especifica el tipo de base de datos.
Name	string	La misma cadena que el nombre del parámetro en la base de datos	Especifica un nombre para el parámetro.

3.1.2 MAPEO DE CLASES A TRAVÉS DE LA HERRAMIENTA SQLMETAL.EXE.

SqlMetal es una herramienta que opera sobre la declaración de líneas de comandos para generar el código y las asignaciones que se utiliza en *LINQ to SQL*, dado que se trabaja sobre líneas de comandos, se ajusta bien a las bases de datos grandes. Si se aplican los diferentes argumentos que se muestran en esta parte, se pueden realizar diferentes acciones, entre las que se incluyen:

- Se puede generar tanto el código fuente como los atributos de asignación (o un archivo de asignación) desde una base de datos.

- Se puede crear un archivo de lenguaje intermedio de marcado de base de datos (.dbml) desde una base de datos, la cuál puede ser personalizada por el programador.
- Desde un archivo .dbml, generar código y atributos de asignación (o un archivo de asignación)

La ruta de ubicación de *SQLMetal* es *c:\Archivos de programa\Microsoft SDKs\Windows\vn.nn\bin* donde *vn.nn* corresponde a la versión v6.0A. Si se trabaja sobre *Visual Studio*, esta herramienta no es accesible, sólo se puede emplear el diseñador relacional de objetos para generar las clases de entidad. Entre las opciones importantes están:

Opciones de conexión:

Opción	Descripción
/server:<nombre>	Especifica el nombre del servidor de base de datos.
/database:<nombre>	Especifica la base de datos del servidor.
/user:<nombre>	Especifica el identificador de usuario de inicio de sesión. El valor predeterminado es autenticación de Windows.
/password:<contraseña>	Especifica la contraseña de inicio de sesión. El valor predeterminado es autenticación de Windows.
/conn:<cadena de conexión>	Especifica la cadena de conexión a bases de datos. No se puede utilizar con las opciones **/server**, **/database**, **/user** o **/password**. No se debe incluir el nombre de archivo en la cadena de conexión sino que puede agregarse el nombre a la línea de comandos como archivo de entrada. Por ejemplo, en la línea siguiente se especifica "c:\base de datos.mdf" como archivo de entrada: **sqlmetal /code:"c:\base de datos.cs" /language: csharp "c:\base de datos.mdf"**.
/timeout:<segundos>	Especifica el valor de tiempo de espera cuando SqlMetal tiene acceso a la base de datos. Valor predeterminado: 0 (es decir, sin límite de tiempo).

Opciones de extracción:

Opción	Descripción
/views	Extrae las vistas de base de datos.
/functions	Extrae las funciones de base de datos.
/sprocs	Extrae los procedimientos almacenados.

Opciones de resultados:

Opción	Descripción
/dbml[:archivo]	Envía el resultado como .dbml. No se puede utilizar con la opción **/map**.
/code[:archivo]	Envía el resultado como código fuente. No se puede utilizar con la opción **/dbml**.
/map[:archivo]	Genera un archivo de asignación XML en lugar de atributos de asignación. No se puede utilizar con la opción **/dbml**.

Otros:

Opción	Descripción
/language:<lenguaje>	Especifica el lenguaje del código fuente. <lenguaje> válido: vb, csharp. Valor predeterminado: depende de la extensión del nombre del archivo de código.
/namespace:<nombre>	Especifica el espacio de nombres del código generado. Valor predeterminado: sin espacio de nombres.
/context:<type>	Especifica el nombre de la clase de contexto de datos. Valor predeterminado: depende del

	nombre de la base de datos.
/entitybase:<type>	Especifica la clase base de las clases de entidad en el código generado. Valor predeterminado: entidades que no tienen clase base.
/serialization:<opción>	Genera las clases serializables. <opción> válida: None, Unidirectional. Valor predeterminado: None.

SqlMetal se basa en dos pasos para obtener los resultados:

- La extracción de los metadatos de la base de datos en un archivo .dbml.
- La compilación de un archivo de resultados de código.

Mediante las opciones apropiadas en la declaración de la línea de comandos, se puede generar código fuente de *Visual Basic* o *C#* o generar un archivo de asignación *XML*. Para extraer los datos (metadatos) de un archivo .mdf de *SQLServer*, debe especificarse el nombre del archivo .mdf después de todas las demás opciones. Si no se especifica */server*, se asume que es *localhost/sqlexpress*. Para especificar un nombre de archivo de entrada, debe agregarse el nombre a la línea de comandos como archivo de entrada. No se permite incluir el nombre de archivo en la cadena de conexión (mediante la opción */conn*).

Ejemplos:
C:\Archivos de programa\Microsoft SDKs\Windows\v6.0A\bin>

1. Crear un archivo *BDEjemplo.dbml* que incluya los metadatos extraídos de una base de datos en *SQL Server Express:*
   sqlmetal /server:.\SQLExpress /database:BDEjemplo /dbml:BDEjemplo.dbml
2. Crear un archivo *mismetadatos.dbml* que incluya los metadatos de SQL extraídos de un archivo .mdf mediante *SQL Server Express*:
   sqlmetal /dbml:mismetadatos.dbml miarchivodb.mdf
3. Crear el código fuente de un archivo de metadatos *BDEjemplo.dbml*:
   sqlmetal /namespace:Linq /code:Linq.cs /language:csharp BDEjemplo.dbml
4. Generar código fuente directamente a partir de los metadatos de *SQL*:
   sqlmetal /server:.\SQLExpress /database: BDEjemplo /namespace:Linq /code:Linq.cs /language:csharp

3.1.3 A TRAVÉS DE ARCHIVOS EXTERNOS XML.

LINQ to SQL admite la asignación externa de archivos, en el que se usa un archivo *XML* de forma independiente para especificar la asignación entre el modelo de datos de una base de datos y el modelo de objetos que va a ser usado por el programador para implementar la lógica del negocio. El modelo de objetos también se puede generar como un archivo *XML* externo a través de la herramienta *SQLMetal*. Con esto se puede cambiar el comportamiento de la aplicación con la modificación del archivo externo sin tener que compilar de nuevo los binarios o ensamblados de una aplicación. Las ventajas de utilizar un archivo externo son las siguientes:

- Se separa el código de la asignación (archivo *xml* externo) del código de la aplicación. Este enfoque reduce el desorden en el código de la aplicación.
- Un archivo de asignación externo se trata de forma similar a un archivo de configuración. Por ejemplo, se puede actualizar el comportamiento de la aplicación aún después de haber liberado los binarios, con el simple hecho de cambiar el archivo de asignación externo.

Nota: El diseñador relacional de objetos en *Visual Studio* no admite la generación de un archivo de asignación externo.

Para ello el archivo de asignación debe ser un archivo *XML* válido y se debe validar contra un archivo de definición de esquema (*.xsd*) de *LINQ to SQL*. Para ello deben considerarse las siguientes reglas:

- El archivo de asignación debe ser un archivo *XML*.
- El archivo de asignación *XML* debe ser válido según el archivo de definición de esquema *XML*.
- Esta asignación invalida una asignación basada en atributos. Esto quiere decir que al utilizar un origen de asignación externo para crear un objeto *DataContext*, el objeto omite todos los atributos de asignación que se han creado en las clases. Este comportamiento es cierto si la clase está incluida en el archivo de asignación externo.
- *LINQ to SQL* no admite la combinación de la asignación basada en atributos y la de archivos externos.
- La asignación externa puede ser específica del proveedor de base de datos. Se puede asignar la misma clase utilizando asignaciones externas independientes para proveedores independientes. Ésta es una característica que la asignación basada en atributos no admite.

El siguiente comando genera un archivo de asignación externo a partir de la base de datos *BDEjemplo*.

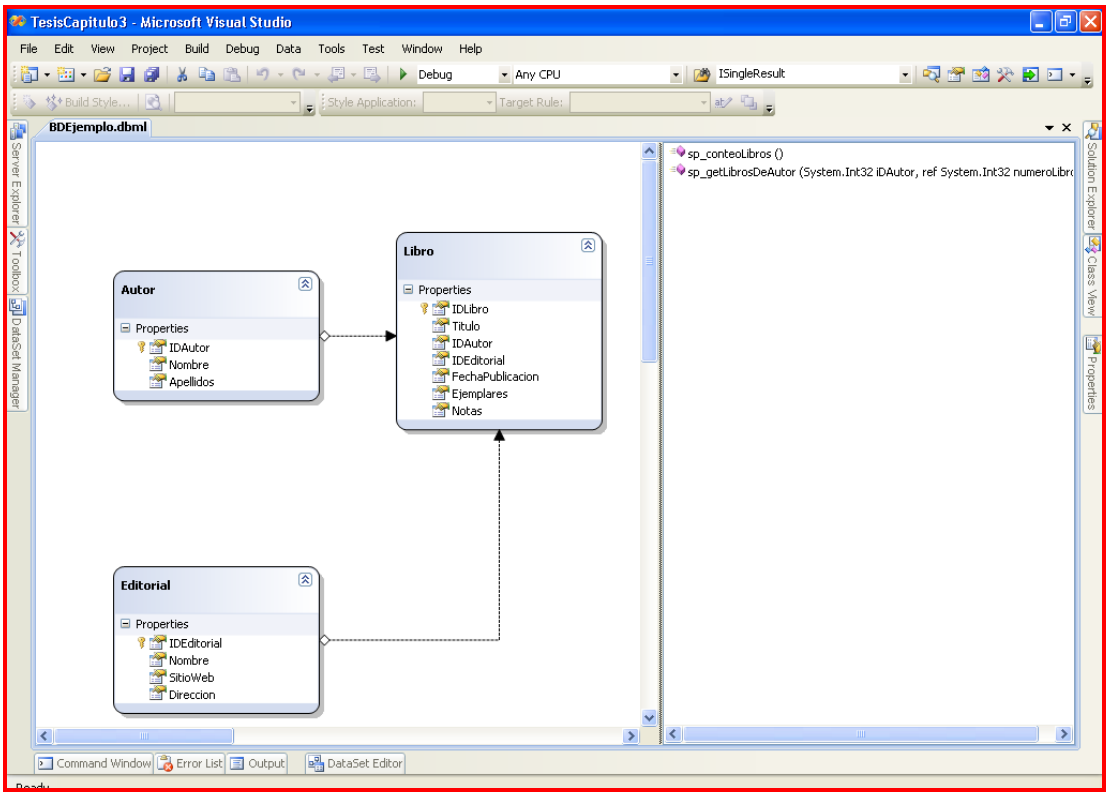
sqlmetal /server:.\sqlexpress /database:BDEjemplo /code:LinqXML /map:BDEjemplo.xml

El siguiente fragmento de un archivo de asignación externo muestra la asignación para la tabla *Autor* de la base de datos *BDEjemplo*. Este fragmento se generó ejecutando *SQLMetal* con la opción */map*.

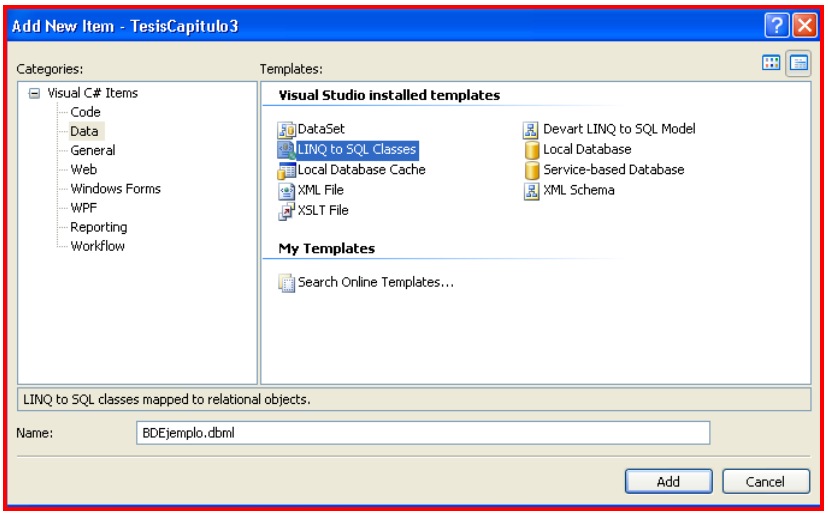
```
<?xml version="1.0" encoding="utf-8"?>
<Database Name="BDEjemplo" xmlns="http://schemas.microsoft.com/linqtosql/mapping/2007">
 <Table Name="dbo.Autor" Member="Autor">
   <Type Name="Autor">
     <Column Name="IDAutor" Member="IDAutor" Storage="_IDAutor" DbType="Int NOT NULL" />
     <Column Name="Nombre" Member="Nombre" Storage="_Nombre"
          DbType="NVarChar(30) NOT NULL" CanBeNull="false" />
     <Column Name="Apellidos" Member="Apellidos" Storage="_Apellidos"
      DbType="NVarChar(50) NOT NULL" CanBeNull="false" />
   </Type>
 </Table>
</Database>
```

3.1.4 A TRAVÉS DEL DISEÑADOR DE OBJETOS RELACIONALES LINQ TO SQL.

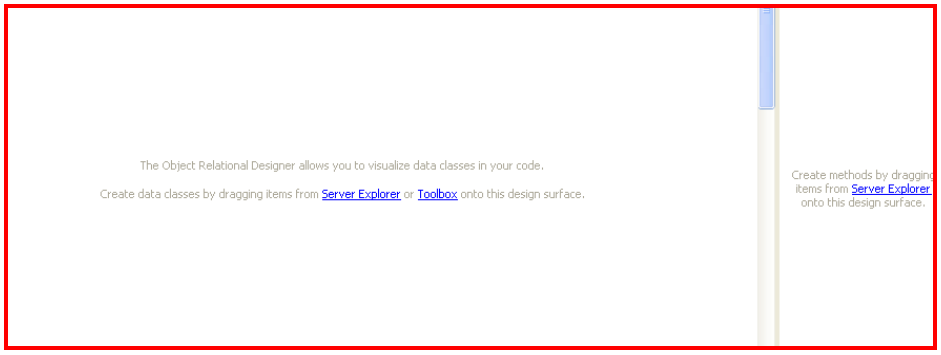
El diseñador relacional de objetos proporciona un área de diseño gráfica para generar clases de entidad y asociaciones (relaciones) de *LINQ to SQL* basadas en los objetos provenientes de una base de datos. Es decir, este diseñador se usa para crear un modelo de objetos para una aplicación que se asigna a los objetos de una base de datos. También genera una clase *DataContext* en el que se garantiza una seguridad de tipos usado para enviar y recibir datos entre las clases del lenguaje de programación y la base de datos. También proporciona la funcionalidad para asignar los procedimientos almacenados y funciones a métodos de *DataContext* con el fin de devolver los datos asociados a estos. Por último, el diseñador relacional de objetos permite crear relaciones de herencia entre las clases de entidad.



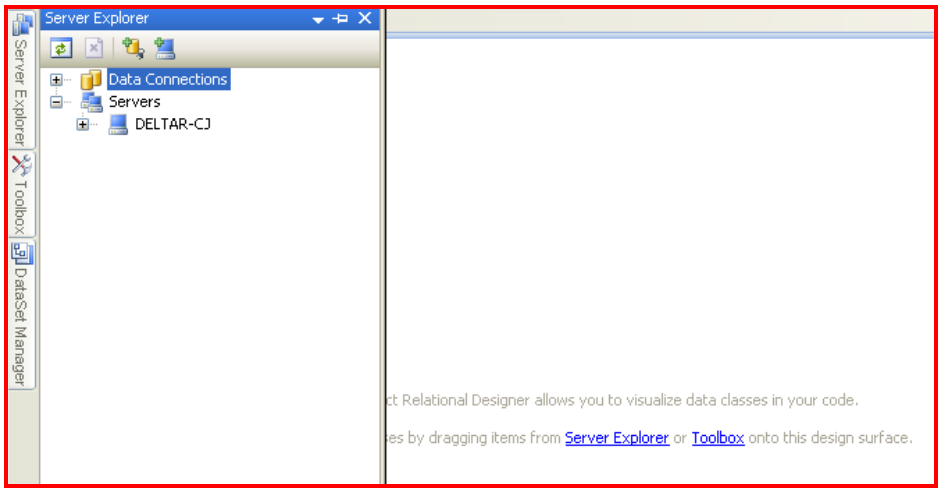
Las clases de entidad se crean y se almacenan en archivos de clases de *LINQ to SQL* (archivos *.dbml*). Este diseñador se inicia al abrir un archivo *.dbml*, al iniciarse, presenta dos áreas distintas en su superficie de diseño: en la parte la izquierda, se encuentra el área asignada a las clases de entidades y en la parte derecha, el área para la asignación de métodos. El panel de entidades es la superficie de diseño principal que muestra las clases de entidad, asociaciones y jerarquías de herencia. El panel de métodos es la superficie de diseño que muestra los métodos *DataContext* que están asignados a procedimientos almacenados y funciones de una base de datos. Para generar el modelo de objetos y el archivo de lenguaje intermedio de marcado de base de datos (*.dbml*) de la base de datos *BDEjemplo*, se empleará el diseñador relacional de objetos incluido en *Visual Studio 2008*. En primera instancia se debe tener un proyecto sobre el lenguaje que este habilitado para *LINQ*, en este caso *C#* (3.0), sobre el *IDE Visual Studio* seleccionar la opción "*Proyecto*" y posteriormente la opción "*Agregar nuevo objeto*", en donde se escogerá la opción "*Clases de LINQ to SQL*", el nombre que se le agregará será *BDEjemplo.dbml* (el mismo de la base de datos):



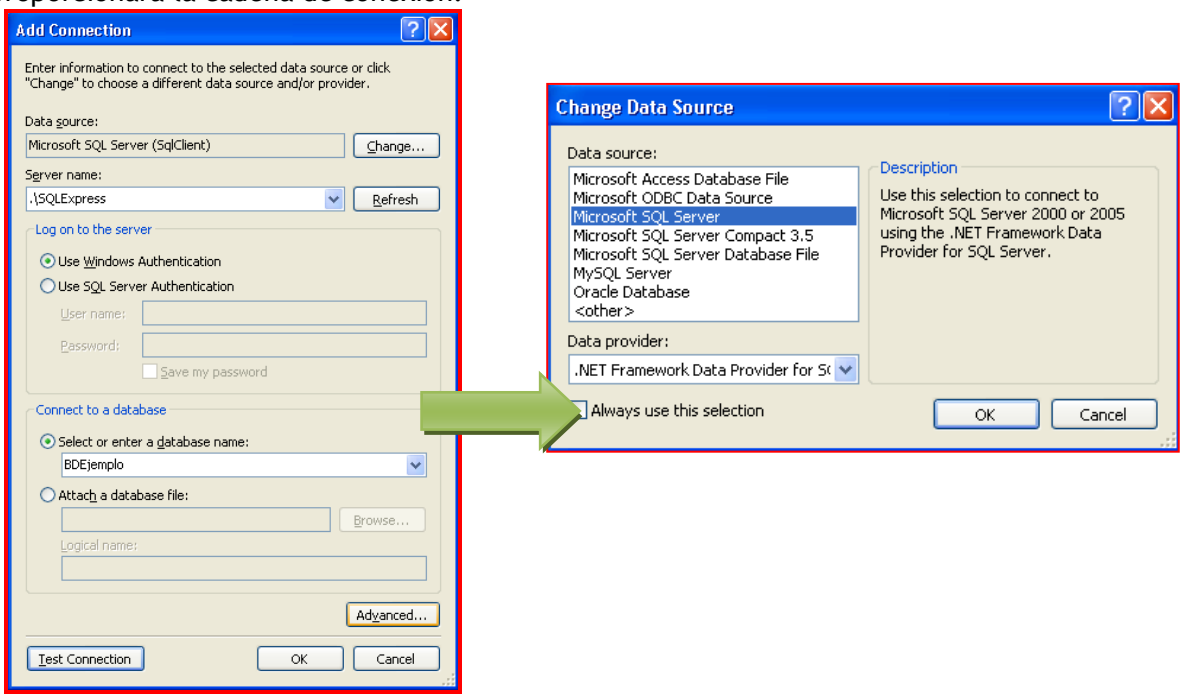
Realizando los pasos anteriores, estará disponible el área del diseñador como en la siguiente imagen, en donde se puede distinguir las dos áreas mencionadas anteriormente: del lado izquierdo se colocarán las tablas y del lado derecho los procedimientos almacenados y/o funciones.



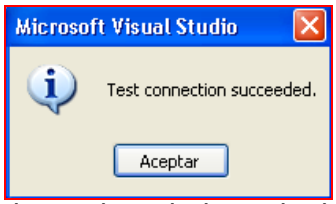
En la parte derecha se nos indica que podemos arrastrar los objetos hacia la superficie a través de explorador de Servidor (*Server Explorer*) o del *ToolBox*, con lo cuál escogeremos la primera opción. Dado que es la primera vez que nos conectaremos a la base de datos *BDEjemplo* contenida en *SQL Server Express Edition 2005* daremos clic derecho sobre Data *Connections* y agregaremos una nueva conexión desde *Visual Studio* hacia la base de datos:



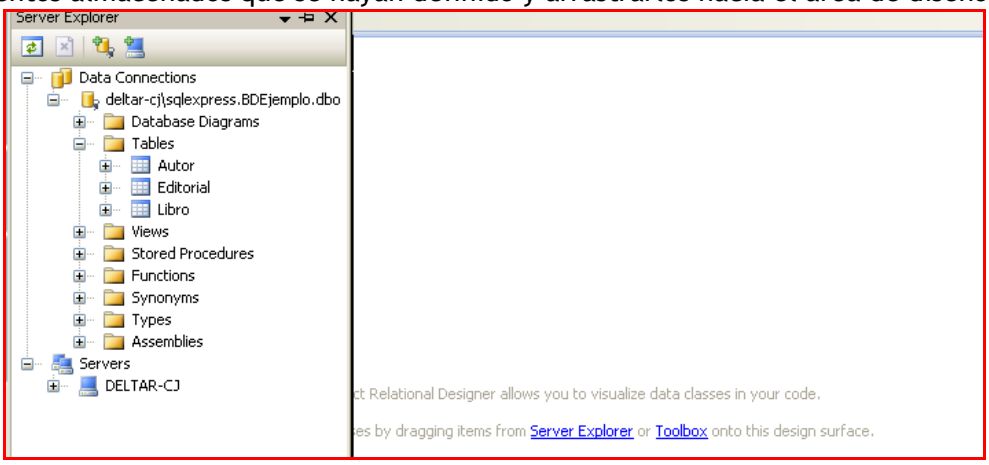
Al seguir el paso anterior, aparecerá una ventana en al que se debe escoger la fuente de datos para la conexión con el servidor y la base de datos, es decir, al proveedor de la conexión hacia el servidor, en este caso se escogerá la opción *Microsoft SQL Server*, el cuál nos proporcionara la cadena de conexión:



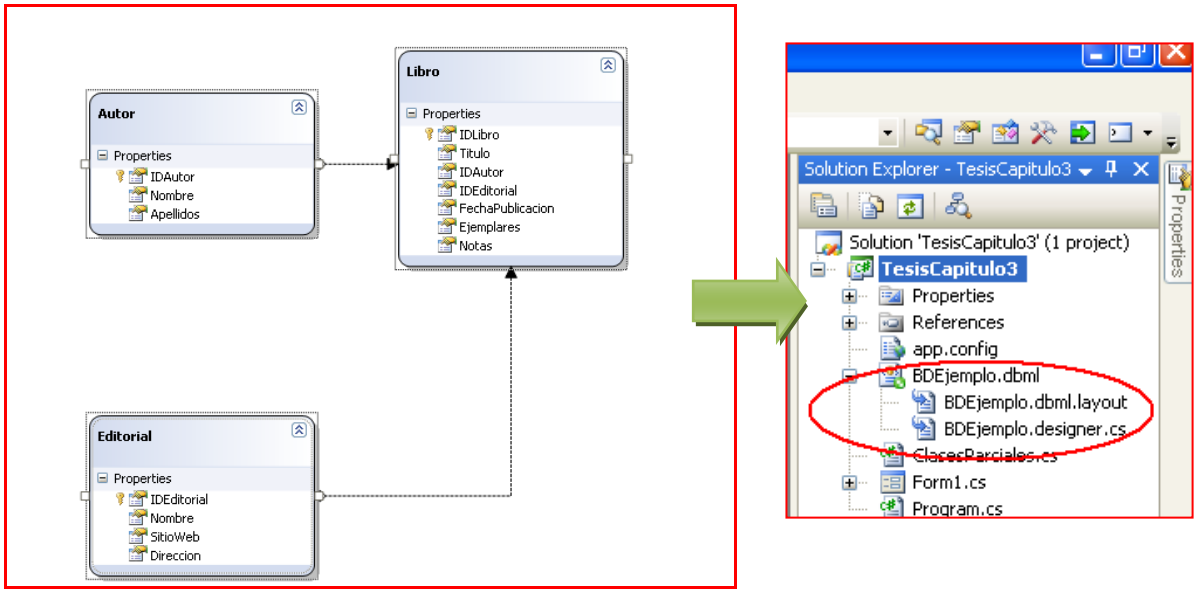
Se puede probar si la conexión creada previamente está correctamente configurada al dar clic sobre el botón "*Probar conexión*" (*Test Connection*), si la conexión es exitosa se mostrará la siguiente imagen, en caso contrario se deberá revisar el procedimiento realizado anteriormente hasta encontrar la falla:



Teniendo la conexión hacia el servidor y la base de datos, se pueden agregar las tablas y procedimientos almacenados que se hayan definido y arrastrarlos hacia el área de diseño:



La herramienta se encargará de generar el modelo de objetos y el archivo *.dbml*, junto con las clases y otros elementos como en la siguiente imagen:



La localización de la cadena de conexión que se creó está en el archivo *app.config* de la aplicación:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
    </configSections>
    <connectionStrings>
        <add name="TesisCapitulo3.Properties.Settings.BDEjemploConnectionString"
            connectionString="Data Source=.\SQLExpress;Initial Catalog=BDEjemplo;Integrated
Security=True"
            providerName="System.Data.SqlClient" />
    </connectionStrings>
</configuration>
```

Con lo anterior, se puede trabajar con la base de datos a través del lenguaje de programación haciendo uso de *LINQ To SQL*.

3.2 MANIPULACIÓN DE LOS DATOS A TRAVÉS DE LAS CONSULTAS.

En los siguientes ejemplos, se usará la base de datos y su respectivo conjunto de clases generadas por el diseñador.

3.2.1 DESARROLLANDO CONSULTAS SOBRE LA BASE DE DATOS.

3.2.1.2 Seleccionar.

El código mostrado abajo usa sintaxis de consulta *LINQ* para regresar una secuencia de objetos *Libro* (*IEnumerable*) provenientes de la base de datos *BDEjemplo*

```
BDEjemploDataContext bd= new BDEjemploDataContext();
var query = from q in bd.Libros select q;
//Mostrar los datos según el criterio del programador
foreach (var item in query)
Console.WriteLine(item.Aut.Nombre+item.Lib.Titulo);
```

En este código, ahora se seleccionan los libros junto con su autor haciendo uso de la cláusula *join*.

```
var query = from libro in bd.Libros
       join autor in bd.Autors
       on libro.IDAutor equals autor.IDAutor
       select new {Lib=libro,Aut=autor};

//Mostrar los datos que queramos
foreach (var item in query)
Console.WriteLine(item.Aut.Nombre+item.Lib.Titulo);
```

3.2.1.3 Insertar.

El código mostrado abajo muestra como crear un libro, posteriormente asociarlo a un autor a través del *IDAutor* y a una editorial ya existente en la base de datos. El libro será insertado a través del método *InsertOnSubmit()*, con lo cuál se agrega la entidad en un estado *pending insert*, esto significa que la entidad agregada no aparecerá en los resultados de consulta de esta tabla hasta que se haya llamado a *SubmitChanges()*. Todos los datos serán guardados en la base de datos a través del método *SubmitChanges()*, de esta forma los objetos regulares (objetos *CLR*) se vuelven persistentes en la base de datos.

```
BDEjemploDataContext bd= new BDEjemploDataContext();

Libro nuevolibro = new Libro() { IDLibro = 6, Titulo = "Nuevo libro adquirido", IDAutor = 2,
IDEditorial = 4, FechaPublicacion = DateTime.Now, Notas = "Notas nuevo libro" };
bd.Libros.InsertOnSubmit(nuevolibro);
bd.SubmitChanges();
```

Para insertar filas en una base de datos, se agregan objetos a la colección *Table<TEntity>* de *LINQ to SQL* asociada y después se envían los cambios a la base de datos. *LINQ to SQL* convierte los cambios en los comandos *INSERT* adecuados de *SQL*.

También se puede insertar un nuevo autor para posteriormente asociarlo a un nuevo libro a través de las llaves foráneas declaradas en la base de datos *BDEjemplo* y de las relaciones creadas por el diseñador (haciendo uso del atributo *Association*) sin hacer uso explícito de la propiedad *IDAutor* de la clase *Libro*:

```
BDEjemploDataContext bd= new BDEjemploDataContext();

Autor nuevoAutor = new Autor()
{IDAutor=7,Nombre="Mauricio",Apellidos="Urrutia Suárez" };

Libro nuevolibro = new Libro()
{ IDLibro = 7, Titulo = "Diseño de circuitos electricos", IDEditorial = 4, FechaPublicacion
= DateTime.Now, Notas = "Notas nuevo libro", Ejemplares = 23,Autor=nuevoAutor};

bd.Autors.InsertOnSubmit(nuevoAutor);
bd.Libros.InsertOnSubmit(nuevolibro);
bd.SubmitChanges();
```

3.2.1.4 Actualizar.

El código mostrado abajo usa sintaxis de consulta *LINQ* junto con un método de extensión (*Single*) para regresar un solo objeto *Libro* (valor *Singleton*) proveniente de la base de datos *BDEjemplo*, actualizar algunas de sus propiedades y posteriormente guardar los cambios en la base de datos:

```
BDEjemploDataContext bd= new BDEjemploDataContext();
Libro query = (from q in bd.Libros where q.IDLibro == 1 select q).Single();
query.Ejemplares = 50;
bd.SubmitChanges();
```

3.2.1.5 Eliminar.

El código mostrado abajo usa la misma sintaxis presentada anteriormente para regresar un solo objeto Libro proveniente de la base de datos *BDEjemplo* y posteriormente eliminarla:

```
BDEjemploDataContext bd= new BDEjemploDataContext();
Libro query = (from q in bd.Libros where q.IDLibro == 7 select q).Single();
bd.Libros.DeleteOnSubmit(query);
bd.SubmitChanges();
```

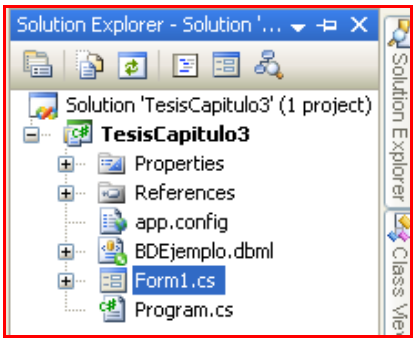
3.2.2 EJECUCIÓN DE PROCEDIMIENTOS ALMACENADOS.

3.2.2.1 Mapeo de procedimientos almacenados.

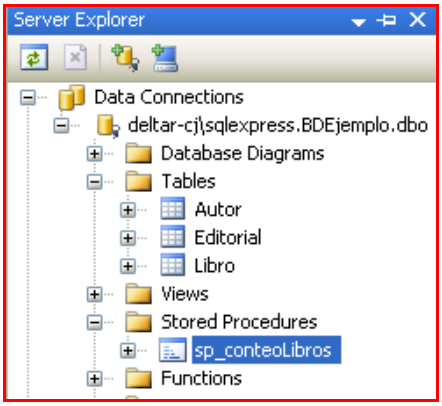
LINQ To SQL también permite trabajar de forma más sencilla con procedimientos almacenados, además de que también pueden ser mapeos hacia métodos que son entendibles por *.NET*. En el siguiente fragmento de código se hace uso de *Transact-Sql* para definir un procedimiento almacenado en *SQL Server* que regresa el número de libros existentes en la base de datos:

```
create procedure sp_conteoLibros
AS
BEGIN
Declare @NumeroLibros int;
SELECT @NumeroLibros=COUNT(*) FROM Libro;
return @NumeroLibros;
END
```

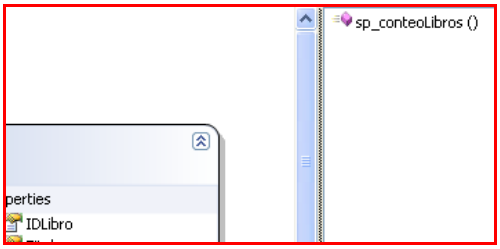
Para poder mapear el procedimiento almacenado definido anteriormente, ir al explorador de solución (*Solution Explorer*) para poder abrir el archivo *.dbml* generado anteriormente (*BDEjemplo.dbml*) y hacer doble clic en él para abrir el diseñador *OR*:



Dado que previamente se realizó el mapeo de las tablas de esta base de datos, la conexión hacia esta última ya se encuentra en el explorador del servidor basta con seleccionar el procedimiento almacenado y arrastrarlo hacia el área derecha de las clases definidas anteriormente:



Aquí el método ya se encuentra en el modelo y en el archivo *.dbml*, bastará con guardar el archivo para hacer permanentes los cambios en el archivo *BDEjemplo.designer.cs* (código *c#*)



En el siguiente fragmento de código solo se muestra el método resultante del mapeo:

```
[Function(Name="dbo.sp_conteoLibros")]
public int sp conteoLibros()
{
IExecuteResult result =
this.ExecuteMethodCall(this,((MethodInfo)(MethodInfo.GetCurrentMethod())));
return ((int)(result.ReturnValue));
}
```

3.2.2.2 Llamada a través del código.

En el siguiente código se muestra cómo llamar al procedimiento almacenado encapsulado en un método normal, primero se hace una instancia del objeto *BDEjemploDataContext* como en los casos anteriores y posteriormente se llama al método *sp_conteoLibros()*, el valor resultante será un valor del tipo *int*:

```
BDEjemploDataContext bd= new BDEjemploDataContext();
int numeroLibros = bd.sp_conteoLibros();
//El valor de numero libros será los que se encuentren en la BD
bd.Dispose();
```

El ejemplo anterior resulta sencillo, ahora se verá cómo devolver varios Libros en un procedimiento almacenado de acuerdo a un parámetro de entrada (en esta caso *IDAutor*). El primer valor devuelto será el conteo de los libros que tiene cierto autor y el segundo serán los

libros que posee el autor. Primero se define el procedimiento almacenado en el servidor *SQL Server* usando *Transact-Sql*:

```
create procedure sp_getLibrosDeAutor(@IDAutor int,@NumeroLibros int OUTPUT)
AS
BEGIN
SELECT @NumeroLibros=COUNT(*) FROM Libro Where IDAutor=@IDAutor;
SELECT * FROM Libro Where IDAutor=@IDAutor;
END
```

Posteriormente se realiza el mapeo siguiendo los pasos descritos anteriormente con lo cuál el resultado del mapeo será el siguiente:

```
[Function(Name="dbo.sp getLibrosDeAutor")]
public ISingleResult<sp_getLibrosDeAutorResult> sp_getLibrosDeAutor([Parameter(Name="IDAutor",
DbType="Int")] System.Nullable<int> iDAutor, [Parameter(Name="NumeroLibros", DbType="Int")] ref
System.Nullable<int> numeroLibros)
{
IExecuteResult result = this.ExecuteMethodCall(this,
((MethodInfo)(MethodInfo.GetCurrentMethod())), iDAutor, numeroLibros);

numeroLibros = ((System.Nullable<int>)(result.GetParameterValue(1)));
return ((ISingleResult<sp_getLibrosDeAutorResult>)(result.ReturnValue));
              }
```

Como se observa, ahora el valor devuelto es una interfaz de nombre *ISingleresult<T>* (espacio de nombres *System.Data.Linq*), donde T es el tipo a devolver. Esta interfaz implementa a *IEnumerable<T>*, por lo que se puede deducir que se devolverá una secuencia del tipo *getLibrosDeAutorResult*, es decir, los libros que posea el autor. Los parámetros que toma este método (precedidos por los atributos *Parameter* descrito anteriormente) se encuentran definidos de tal forma que sean entendibles por *.NET* (tipos *SQL* -> tipos *.NET*). El primer parámetro es del tipo *Nullable<int>* ya que en *SQL Server* se pueden admitir valores int o nulos (*@IDAutor*), el segundo parámetro es el segundo parámetro de salida definido en el procedimiento almacenado (*@NumeroLibros*) el cuál contiene el número de libros del autor la cuál esta precedida por la palabra ref, lo que significa que se hace el paso de un valor por referencia.

Como se mencionó en el párrafo anterior, se devuelve una secuencia de valores *getLibrosDeAutorResult*, esta clase también es creada por el diseñador, la cuál se muestra a continuación y se encuentra de forma resumida, por lo que solo se muestran los campos con los que cuenta esta clase:

```
public partial class sp_getLibrosDeAutorResult{
   private int _IDLibro;
   private string _Titulo;
   private int _IDAutor;
   private int _IDEditorial;
   private System.DateTime _FechaPublicacion;
   private System.Nullable<int>  Ejemplares;
   private string _Notas;
}
```

Como se observa, los campos mostrados son los mismos que se usan en la clase *Libro*, por lo que se puede concluir que el valor devuelto será un objeto del tipo *Libro* y con la práctica aunada a la experiencia extendida, el programador puede modificarlo de tal forma que en vez de devolver una secuencia de sp_*getLibrosDeAutorResult* fácilmente se puede hacer uso de la clase *Libro* y reutilizar el código, con lo que el resultado sería el siguiente:

```
[Function(Name="dbo.sp_getLibrosDeAutor")]
public ISingleResult<Libro> (...){
    ...
    return ((ISingleResult<Libro>)(result.ReturnValue));
}
```

Para poder llamar al procedimiento almacenado a través del código, se instancia la clase *BDEjemploDataContext*. Para obtener el valor del segundo parámetro del procedimiento almacenado (*@NumeroLibros*) primero se declara una variable *Nullable<int>* y se inicializa, para posteriormente ser pasada por referencia en el método sp_*getLibrosDeAutor* con la palabra clave *ref*, a partir de allí el valor de esta variable tendrá el numero de libros que tiene un autor. Una variable "*resultado*" estará asignada al valor devuelto por el método y contendrá los libros del autor:

```
BDEjemploDataContext bd = new BDEjemploDataContext();
Nullable<int> numeroLibros =null;
ISingleResult<sp_getLibrosDeAutorResult> resultado =
                    bd.sp_getLibrosDeAutor(2, ref numeroLibros);
//numeroLibros contiene ahora el numero de los libros del autor con id=2
foreach(sp_getLibrosDeAutorResult valor in resultado) Console.WriteLine(valor.Titulo);
//En el ciclo anterior se muestran los libros que tiene un autor con id=2
```

CAPITULO IV. CONCEPTOS AVANZADOS SOBRE EL MANEJO DE BASE DE DATOS RELACIONALES CON LINQ TO SQL.

4.1 CLASE DATACONTEXT.

Como se observó en el capítulo anterior, las consultas en *LINQ to SQL* que se ejecutan sobre la base de datos, se hace uso de instancias de *DataContext*, esta clase es el origen de todas las entidades asignadas en una conexión de base de datos. Realiza un seguimiento de los cambios realizados en todas las entidades recuperadas de la base de datos, y mantiene una "memoria caché de identidad", con lo que se garantiza que las entidades u objetos que se recuperan más de una vez de la base de datos, se representen utilizando la misma instancia de objeto. En general, una instancia de *DataContext* está diseñada para que dure una "unidad de trabajo", sea cual sea la definición de ese término en la aplicación. Una aplicación *LINQ to SQL* típica crea instancias de *DataContext* que representan un conjunto de operaciones congruentes sobre la base de datos. En el capítulo anterior se mostró como crear un modelo de objetos en *Visual Studio 2008* asignada a una base de datos como *SQL Server Express Edition 2005*. Al arrastrar los elementos de la herramienta de trabajo *Server Explorer* (*Explorador de Servidores*) hacia el área de diseño, se generan las clases de mapeo, estas son clases que representan los objetos de base de datos. En el siguiente fragmento de código, se muestra la clase generada en su forma más simple como forma de ejemplo:

```
[System.Data.Linq.Mapping.DatabaseAttribute(Name="BDEjemplo")]
public partial class BDEjemploDataContext : System.Data.Linq.DataContext {

public BDEjemploDataContext() :
base(global::TesisCapitulo3.Properties.Settings.Default.BDEjemploConnectionString,
mappingSource) { }

public System.Data.Linq.Table<Autor> Autors
{get{return this.GetTable<Autor>();}}

public System.Data.Linq.Table<Libro> Libros
{get{return this.GetTable<Libro>();}}

public System.Data.Linq.Table<Editorial> Editorials
{get{return this.GetTable<Editorial>();}}

[Table(Name="dbo.Autor")]
public partial class Autor : INotifyPropertyChanging, INotifyPropertyChanged {
\\Los campos y propiedades de esta clase corresponden tanto en numero como nombre a
\\los que se encuentran definidos en la base de datos, el tipo que tenga en la base de
\\datos será su similar en .NET
}

[Table(Name="dbo.Libro")]
public partial class Libro : INotifyPropertyChanging, INotifyPropertyChanged {
\\Los campos y propiedades de esta clase corresponden tanto en numero como nombre a
\\los que se encuentran definidos en la base de datos, el tipo que tenga en la base de
\\datos será su similar en .NET
}

[Table(Name="dbo.Editorial")]
public partial class Editorial : INotifyPropertyChanging, INotifyPropertyChanged {
\\Los campos y propiedades de esta clase corresponden tanto en numero como nombre a
\\los que se encuentran definidos en la base de datos, el tipo de datos que tenga
\\en la base de datos será su similar en .NET
}
```

Como se puede observar, se tienen las siguientes clases:

- Una clase *DataContext*. Que representa el contexto de la conexión.
- Una clase por cada tabla que se haya seleccionado. Cada clase tendrá una propiedad por cada campo de la tabla a la que representa.

La clase *DataContext* actúa como una vía entre una base de datos de *SQL Server* y las clases de entidad de *LINQ to SQL* asignadas a esa base de datos. Esta misma clase contiene la información de la cadena de conexión y los métodos para realizar la conexión a una base de datos y manipular los datos de la base de datos:

```
public BDEjemploDataContext() :
base(global::TesisCapitulo3.Properties.Settings.Default.BDEjemploConnectionString,
 mappingSource)
{
        OnCreated();
}
```

Uno de los constructores (por default) de la clase que hereda de *DataContext* hace una llamada al constructor de la clase base (*DataContext*), en la que se le dan como parámetros la cadena de conexión hacia la base de datos y la segunda la fuente de la cual se va a mapear la información (en este caso esta clase), por lo que al instanciar esta clase haciendo uso de su constructor por default en realidad se llama al constructor de *DataContext*.

4.2 DEFINICIONES PARCIALES PARA LA LÓGICA DEL NEGOCIO PERSONALIZADA.

4.2.1 CLASES PARCIALES.

Al observar las clases anteriores se nota que todas tienen un cosa en común en el principio de su definición, se hace uso de la palabra *partial* antes de seguir con la palabra clave para definir una clase, *class*. Con esto se crea una clase parcial (este concepto fue introducido en *.NET FarmeWork 2.0*), con lo que es posible dividir la definición de una clase, estructura o interfaz en dos o más archivos de código fuente. Cada uno de estos archivos contiene una parte de la definición de clase y todas las partes se combinan cuando se compila la aplicación. La división de la definición de una clase resulta útil en escenarios como:

- En proyectos grandes, el expandir una clase en archivos independientes permite que varios programadores trabajen simultáneamente con ella.
- Con un código que se genera de forma automática, se le puede agregar más comportamientos a una clase definida sin la necesidad de crear el archivo de código fuente de nuevo. En este caso, *Visual Studio* utiliza usa clases parciales al crear formularios *Windows Forms*, código de contenedor para un servicio web, etc.

Como se mencionó anteriormente, el uso de la palabra clave *partial* indica que otras partes de la clase, estructura o interfaz se pueden definir dentro del espacio de nombres, sin embargo se deben tomar en cuenta consideraciones al momento de querer hacer su uso:

1. El modificador *partial* no está disponible en declaraciones de delegado o enumeración.
2. Todas las partes deben utilizar la palabra clave *partial*.
3. Todas las partes deben estar disponibles en tiempo de compilación para formar el tipo final.
4. Todas las partes deben tener la misma accesibilidad, ya sea *public*, *private*, etc.
5. Si alguna de las partes se declara abstracta, todo el tipo se considera abstracto.
6. Si alguna de las partes se declara sellada, todo el tipo se considera sellado.
7. Si alguna de las partes declara un tipo base, todo el tipo hereda esa clase

Todas las partes en las que se declara una clase base deben concordar, pero las partes que omiten una clase base heredan igualmente el tipo base. Las partes pueden especificar diferentes interfaces, pero el tipo final implementa todas las interfaces mostradas por todas las

declaraciones parciales. Cualquier miembro de clase, estructura o interfaz declarados en una definición parcial está disponible para todas las demás partes. Cuando se habla de un tipo final, quiere decir que es el resultado de la combinación de todas las partes que tienen definiciones parciales en tiempo de compilación (en tiempo de compilación, se combinan los atributos de definiciones de tipo parcial). Por ejemplo, las siguientes declaraciones:

```
[System.SerializableAttribute]
partial class ClaseParcial { }

[System.ObsoleteAttribute]
partial class ClaseParcial { }
```

Equivalen a:

```
[System.SerializableAttribute]
[System.ObsoleteAttribute]
class ClaseParcial { }
```

Para ejemplificar el uso de las clases parciales, se tiene la clase parcial Autor que *generó* Visual *Studio 2008* al mapear la base de datos *BDEjemplo*. Esta clase tiene los siguientes campos y propiedades:

```
[Table(Name="dbo.Autor")]
public partial class Autor : INotifyPropertyChanging, INotifyPropertyChanged
{
private int _IDAutor;
private string  Nombre;
private string _Apellidos;
private EntitySet<Libro> _Libros;

    partial void OnLoaded();
    partial void OnValidate(System.Data.Linq.ChangeAction action);
    partial void OnCreated();
    partial void OnIDAutorChanging(int value);
    partial void OnIDAutorChanged();
    partial void OnNombreChanging(string value);
    partial void OnNombreChanged();
    partial void OnApellidosChanging(string value);
    partial void OnApellidosChanged();

...

[Column(Storage="_IDAutor", DbType="Int NOT NULL", IsPrimaryKey=true)]
public int IDAutor
{
get{return this._IDAutor;}
set{
        if ((this._IDAutor != value)) {
        this.OnIDAutorChanging(value);
        this.SendPropertyChanging();
        this._IDAutor = value;
        this.SendPropertyChanged("IDAutor");
        this.OnIDAutorChanged();
        }
}
}

[Column(Storage="_Nombre", DbType="NVarChar(30) NOT NULL", CanBeNull=false)]
public string Nombre
{
get{return this. Nombre;}
set{
        if ((this._Nombre != value)) {
        this.OnNombreChanging(value);
        this.SendPropertyChanging();
        this._Nombre = value;
        this.SendPropertyChanged("Nombre");
        this.OnNombreChanged();
```

```
        }
}
}

[Column(Storage="_Apellidos", DbType="NVarChar(50) NOT NULL", CanBeNull=false)]
public string Apellidos
{
get{return this._Apellidos;}
set{
        if ((this._Apellidos != value)) {
        this.OnApellidosChanging(value);
        this.SendPropertyChanging();
        this. Apellidos = value;
        this.SendPropertyChanged("Apellidos");
        this.OnApellidosChanged();
        }
}
}

[Association(Name="Autor_Libro", Storage="_Libros", OtherKey="IDAutor")]
public EntitySet<Libro> Libros {
get{return this._Libros;}
set{
        this._Libros.Assign(value);
        }
}

private void attach_Libros(Libro entity) {
        this.SendPropertyChanging();
        entity.Autor = this;
}

private void detach_Libros(Libro entity) {
        this.SendPropertyChanging();
        entity.Autor = null;
        }
}
}
```

Ahora se procede a agregar un archivo de clase (*.cs*) para código *c#* de forma separada, para ello solo dar clic en el menú *Proyecto* (*Project*) y en la opción Agregar clase (*Add Class*), por lo que se definirá la misma clase parcial y se colocará una nueva propiedad que devuelve el nombre completo del libro, concatenando los campos _Nombre y _Apellidos del Autor (ambas son del tipo *string*):

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

namespace ...
{
    public partial class Autor
    {
        public string NombreCompleto {
            get { return this._Nombre+" "+this._Apellidos;}
        }
    }
}
```

Dado que esta declaración hace que esta nueva propiedad forme parte de la clase parcial *Autor*, se procede de forma natural a realizar una consulta en *LINQ* para devolver todos los autores de la base de datos y observar cómo la nueva propiedad se encuentra dentro del contexto de la clase:

```
BDEjemploDataContext bd = new BDEjemploDataContext();
var query = from q in bd.Autors select q;
```

```
foreach (var item in query)
//En este paso los valores de esta propiedad pueden ser desplegados
/o procesados
   Console.Write(item.NombreCompleto);
```

Las clases parciales no están limitadas para agregar simples campos. También se pueden incluir más funcionalidades que esa, por ejemplo se puede extender el funcionamiento de la clase al implementar interfaces en diferentes archivos separados, agregar nuevos métodos, etc.

4.2.2 Métodos parciales.

Otra característica importante que agrega más funcionalidad a una entidad *DataContext* y que puede ser definida para la lógica del negocio son los métodos parciales. En el siguiente fragmento de código se observa que se encuentran métodos que en su definición tienen la palabra clave *partial* y las cuáles son generadas de forma automática en la generación del mapeo.

```
[Table(Name="dbo.Autor")]
public partial class Autor : INotifyPropertyChanging, INotifyPropertyChanged
{
...

    //Definiciones de metodos parciales
    partial void OnLoaded();
    partial void OnValidate(System.Data.Linq.ChangeAction action);
    partial void OnCreated();
    partial void OnIDAutorChanging(int value);
    partial void OnIDAutorChanged();
    partial void OnNombreChanging(string value);
    partial void OnNombreChanged();
    partial void OnApellidosChanging(string value);
    partial void OnApellidosChanged();
...
}
```

Y en c/u de las propiedades, según el tipo y nombre de esta se hace la llamada a estos métodos, por ejemplo en la propiedad Nombre se llama a los métodos *OnNombreChanging* y *OnNombreChanged*:

```
public string Nombre
{
get{return this._Nombre;}
set{
        if ((this._Nombre != value))
        {
        this.OnNombreChanging(value); //Llamada a uno de los métodos parciales
        this.SendPropertyChanging();
        this._Nombre = value;
        this.SendPropertyChanged("Nombre");
        this.OnNombreChanged();//Llamada a uno de los métodos parciales
        }
}
}
```

Así, por ejemplo, para poder realizar operaciones de afinamiento o tener un mejor control sobre lo que se está realizando en el cambio de la propiedad, puede establecerse la misma clase parcial en la que se define la funcionalidad de los métodos parciales, como en el siguiente ejemplo (recordar que esta nueva definición se encuentra en un archivo de código fuente *.cs* por separado):

```
namespace ...
{
    public partial class Autor
    {
        partial void OnNombreChanging(string value)
```

```
        {
//La funcion del metodo queda a disposicion del programador o de las necesidades
//de la lógica del negocio
            Console.WriteLine("Nombre del autor " + this._Nombre+" cambiando a " + value);
        }
        partial void OnNombreChanged()
        {
//La funcion del metodo queda a disposicion del programador o de las necesidades
//de la lógica del negocio
            Console.WriteLine("Nombre del autor cambiado: "+ this._Nombre);
        }
    }
}
```

La implementación en el código centrado en la lógica del negocio queda de la siguiente forma:

```
BDEjemploDataContext bd = new BDEjemploDataContext();
var query = (from q in bd.Autors where q.IDAutor ==2 select q).Single();
query.Nombre="Nuevo Nombre";
//El resultado es el siguiente:
Nombre del autor Ana Maria cambiando a: Nuevo Nombre
Nombre del autor cambiado: Nuevo Nombre
```

Si implementamos el método en nuestro código, el compilador agregará la funcionalidad. En caso contrario, si no lo implementamos, el compilador removerá el método vacío.

4.3 EJECUCIÓN DE PROCEDIMIENTOS ALMACENADOS CON DIFERENTES CONJUNTOS DE RESULTADOS (RESULTSET).

En el capítulo anterior se mostró como realizar el mapeo de un procedimiento almacenado y cómo llamarlo en el código. Hay dos tipos distintos de métodos de *DataContext*:

1. Métodos que devuelven uno o varios conjuntos de resultados. Este tipo de método se usa cuando en la aplicación se ejecutan los procedimientos almacenados y funciones de la base de datos y devolver los resultados, para ello se usan las interfaces *ISingleResult* e *IMultipleResults* en el espacio de nombres *System.Data.Linq*.
2. Métodos que no devuelven conjuntos de resultados tales como operaciones del tipo *Insert*, *Update* y *Delete* para una clase de entidad en específico. Este tipo de método se usa cuando en la aplicación se ejecutan procedimientos almacenados en lugar de usar el comportamiento predeterminado de *LINQ to SQL* para guardar los datos modificados entre una clase de entidad y la base de datos, a través de los métodos como *InsertOnSubmit*, *InsertAllOnSubmit*, *SubmitChanges*, etc.

Retomando el ejemplo en el que se creo el procedimiento almacenado *sp_getLibrosDeAutor* en el capítulo 3 en la sección 3.2.2.2 (*Llamada a través del código*):

```
create procedure sp_getLibrosDeAutor(@IDAutor int,@NumeroLibros int OUTPUT)
AS
BEGIN
SELECT @NumeroLibros=COUNT(*) FROM Libro Where IDAutor=@IDAutor;
SELECT * FROM Libro Where IDAutor=@IDAutor;
END
```

Se mapeo:

```
[Function(Name="dbo.sp getLibrosDeAutor")]
public ISingleResult<sp_getLibrosDeAutorResult> sp_getLibrosDeAutor([Parameter(Name="IDAutor",
DbType="Int")] System.Nullable<int> iDAutor, [Parameter(Name="NumeroLibros", DbType="Int")] ref
System.Nullable<int> numeroLibros)
{
IExecuteResult result = this.ExecuteMethodCall(this,
((MethodInfo)(MethodInfo.GetCurrentMethod())), iDAutor, numeroLibros);
```

```
numeroLibros = ((System.Nullable<int>)(result.GetParameterValue(1)));
return ((ISingleResult<sp_getLibrosDeAutorResult>)(result.ReturnValue));
            }
```

Y se llamó a través del código:

```
BDEjemploDataContext bd = new BDEjemploDataContext();
Nullable<int> numeroLibros =null;
ISingleResult<sp_getLibrosDeAutorResult> resultado =
                    bd.sp getLibrosDeAutor(2, ref numeroLibros);

//numeroLibros contiene ahora el numero de los libros del autor con id=2

foreach(sp_getLibrosDeAutorResult valor in resultado) MessageBox.Show(valor.Titulo);
//En el ciclo anterior se muestran los libros que tiene un autor con id=2
```

Resulta que este método cae dentro de la definición 1, devuelve un solo conjunto de resultados (*ISingleResults*), pero el problema surge cuando se tiene que devolver más de un conjunto de resultados. Para ello, ha de definirse otro procedimiento almacenado, que tenga ese comportamiento en la base de datos *BDEjemplo* contenida en *SQL Server Express Edition 2005*:

```
CREATE PROCEDURE sp_getLibrosyAutores
AS
BEGIN
SELECT * FROM Libro;
SELECT * FROM Autor;
END
```

Como se puede ver, en este procedimiento almacenado ahora se devuelven 2 conjuntos de resultados. Siguiendo el mismo procedimiento para mapear el procedimiento almacenado hacia nuestro proyecto en *C#* se tiene el siguiente resultado:

```
[Function(Name="dbo.sp getLibrosyAutores")]
public ISingleResult<sp_getLibrosyAutoresResult> sp_getLibrosyAutores(){
IExecuteResult result = this.ExecuteMethodCall(this,
                                ((MethodInfo)(MethodInfo.GetCurrentMethod())));
return ((ISingleResult<sp getLibrosyAutoresResult>)(result.ReturnValue));
}
```

Además de generarse el fragmento de código anterior, como se mencionó en el capítulo 3 en la sección de procedimientos almacenados, también se genera la clase *sp_getLibrosyAutoresResult* que soporta los resultados devueltos y cuya definicion es la siguiente:

```
public partial class sp getLibrosyAutoresResult{
        private int _IDLibro;
        private string _Titulo;
        private int _IDAutor;
        private int _IDEditorial;
        private System.DateTime _FechaPublicacion;
        private System.Nullable<int> _Ejemplares;
        private string _Notas;

        public sp_getLibrosyAutoresResult(){  }


        [Column(Storage=" Titulo", DbType="NVarChar(50)")]
        public int Titulo {
         get { return this._Titulo;}
          set{
                if ((this._Titulo != value))
                {this. Titulo = value;}
          }
        }

        [Column(Storage="_IDAutor", DbType="Int NOT NULL")]
        public int IDAutor {
         get{return this._IDAutor;}
         set{
```

```
                if ((this._IDAutor != value))
                {this._IDAutor = value;}
          }
        }

        [Column(Storage="_IDEditorial", DbType="Int NOT NULL")]
        public int IDEditorial {
          get{return this._IDEditorial;}
          set{
                if ((this._IDEditorial != value))
                {this._IDEditorial = value; }
          }
        }

        [Column(Storage="_FechaPublicacion", DbType="DateTime NOT NULL")]
        public System.DateTime FechaPublicacion {
         get{ return this._FechaPublicacion; }
         set {
                if ((this._FechaPublicacion != value))
                {this._FechaPublicacion = value; }
          }
        }

        [Column(Storage="_Ejemplares", DbType="Int")]
        public System.Nullable<int> Ejemplares {
          get {return this._Ejemplares; }
          set {
                if ((this._Ejemplares != value))
                {this._Ejemplares = value; }
          }
        }

        [Column(Storage="_Notas", DbType="NVarChar(200)")]
        public string Notas {
          get{return this._Notas; }
          set {
                if ((this._Notas != value))
                {this._Notas = value; }
          }
        }
}
```

El valor devuelto del método *sp_getLibrosyAutores*, sin embargo, es un tipo *ISingleResult*, dado que esta interfaz representa el resultado de una función asignada que tiene una secuencia de retorno única (una secuencia de valores de un solo tipo) por lo que contrasta con el procedimiento almacenado, en el que se devuelven dos conjuntos de resultados de dos tipos: *Libro* y *Autor*. Con esto último se puede aseverar que el mapeo de procedimientos almacenados de este tipo no es el fuerte de *LINQ to SQL* ni de la herramienta diseñador relacional de objetos. Por lo que el trabajo de modificar o crear (desde el punto de visto donde sea tomado) este método le toca al programador.

Dado que se requiere obtener más de una secuencia de resultados la interfaz *ISingleResult* resulta inútil, para ello ha de apoyarse en la interfaz IMultipleResults, esta interfaz representa los resultados de consultas o procedimientos asignados con secuencias de devolución diferentes (multiples secuencias de valores), esta interfaz se encuentra en el mismo espacio de nombre que *ISingleResult*, esta interfaz se encuentra definida de la siguiente forma:

```
public interface IMultipleResults : IFunctionResult, IDisposable
{
   //se devuelve una secuencia de elementos TElement, puesto que es un IEnumerable<TElement>
    IEnumerable<TElement> GetResult<TElement>();
}
```

Puesto que esta interfaz tiene este comportamiento, se puede usar para poder devolver este tipo en el método generado. Por lo que, al modificar el método generado para que el valor devuelto sea del tipo *IMultipleResults*, queda de la siguiente forma:

```
public IMultipleResults sp_getLibrosyAutores()
{
IExecuteResult result = this.ExecuteMethodCall
                        (this, ((MethodInfo)(MethodInfo.GetCurrentMethod())));
return ((IMultipleResults)(result.ReturnValue));
}
```

Ahora el metodo puede devolver una secuencia de valores diferentes a través de un *IExecuteResult*, esta proporciona acceso al valor devuelto, o a los resultados de la ejecución de una consulta, que a su vez son un *IMultipleResults*. Sin embargo, al modificar el metodo no se sabe exactamente que tipos o secuencias de valores ha de devolverse, es decir, que se pueden devolver varias secuencias de valores de diferente tipo, pero la pregunta es: ¿de qué tipo son las secuencias que se devuelven?.

Se sabe que el procedimiento almacenado devuelve una secuencia de libros y otra secuencia de autores, por lo que deben crearse las clases en *c#* que soporten el tipo libro y autor. Como anteriormente se había creado un procedimiento alamacenado en el capítulo 3, que devolvía tanto el numero de libros (*int*) que tenia un autor como los libros que tenia (*ISingleResult<sp_getLibrosDeAutorResult>*), se generó automáticamente la clase *sp_getLibrosDeAutorResult* que soportaba ese tipo de resultado y los campos y propiedades que contiene, son los mismos en numero de columnas que devolvia el procedimiento. Por lo que se puede hacer reuso de esa clase para el caso de la primera secuencia devuelta: la de libros, en caso contrario de no haber realizado un mapeo de un procedimiento almacenado con tales caracteristicas, se puede hacer uso de la clase que genera el diseñador relacional de objetos (*sp_getLibrosyAutoresResult*). Faltaría definir otra clase que soporte la segunda secuencia de valores: la de autores. Para este caso ha de definirse una nueva clase que soporte esta clase de tipo, para cuyo caso ha de llamarse *sp_getAutoresResult* y su definición queda establecida de la siguiente manera:

```
public partial class sp_getAutoresResult
{
   private int  IDAutor;
   private string _Nombre;
   private string _Apellidos;
   public sp_getAutoresResult()
   { }

[Column(Storage = "_IDAutor", DbType = "Int NOT NULL")]
public int IDAutor {
        get { return this._IDAutor; }
        set {
               if ((this._IDAutor != value)){
                   this._IDAutor = value;
                }
        }
}
[Column(Storage = "_Nombre", DbType = "NVarChar(30) NOT NULL", CanBeNull = false)]
public string Nombre {
        get { return this. Nombre; }
        set {
               if ((this._Nombre != value)) {
                   this._Nombre = value;
                }
        }
}
[Column(Storage = "_Apellidos", DbType = "NVarChar(50)")]
public string Apellidos {
        get { return this._Apellidos; }
```

```
        set {
               if ((this._Apellidos != value)) {
                   this._Apellidos = value;
                }
        }
}
```

Al prestar un poco más de atención, el numero de propiedades también corresponden al numero de columnas devueltas por el procedimiento almacenado. Para terminar faltaría definir en el método estos tipos, para ello se hace uso del atributo *ResultType*, que se utiliza en procedimientos almacenados que devuelven un tipo *IMultipleResults* y en la cuál se colocará el tipo de resultado que devuelve el procedimiento almacenado (haciendo uso de la palabra clave typeof, ya que lo que se requiere es un *Type*) y que se crearon en *.NET* por parte del programador, es decir que la definicion final del método queda de la siguiente forma:

```
[Function(Name="dbo.sp_getLibrosyAutores")]
[ResultType(typeof(sp_getLibrosDeAutorResult))]
[ResultType(typeof(sp_getAutoresResult))]
public IMultipleResults sp_getLibrosyAutores() {
        IExecuteResult result = this.ExecuteMethodCall
                                (this,((MethodInfo)(MethodInfo.GetCurrentMethod())));
return ((IMultipleResults)(result.ReturnValue));
}
```

Ahora solo basta con llamar al método através del código, para ello se muestra el siguiente fragmento de código:

```
BDEjemploDataContext bd = new BDEjemploDataContext();
//se llama al método y se asigna el valor devuelto a un IMultipleResults
IMultipleResults res = bd.sp getLibrosyAutores();

//del objeto IMultipleResults se obtienen c/u de las 2 secuencias:

//secuencia no. 1 : IEnumerable<sp_getLibrosDeAutorResult>
IEnumerable<sp_getLibrosDeAutorResult> resul1 = res.GetResult<sp_getLibrosDeAutorResult>();

//secuencia no. 2 : IEnumerable<sp_getAutoresResult>
IEnumerable<sp getAutoresResult> resul2 = res.GetResult<sp getAutoresResult>();

//se muestran o procesan las 2 secuencias
foreach (sp_getLibrosDeAutorResult libro in resul1) Console.WriteLine(libro.Titulo);
foreach (sp_getAutoresResult autor in resul2) Console.WriteLine(autor.Nombre);
```

En el código anterior lo que se realiza es la obtención del resultado del método *sp_getLibrosyAutores*, el cuál es un tipo *IMultipleResults*, a partir de este tipo se obtienen las secuencias a través del método *GetResult<TElement>()* definida en esta interfaz, esto nos devolverá las secuencias de valores.

4.4 USO DE TRANSACCIONES.

LINQ to SQL porporciona tres modelos principales para administrar las transacciones, de los cuales en dos de estos se hace uso de DataContext.

4.4.1 TRANSACCIÓN IMPLÍCITA.

En esta opción, *DataContext* crea y lista una transaccion cuando sea llamado el método *SubmitChanges*. En este método, dependiendo de la opcion *ConflictMode* seleccionada, deshará los cambios realizados de forma automática. Para este caso *LINQ to SQL* comprueba si la llamada a *SubmiteChanges* se encuentra dentro de una transaccion (*Transaction*) o si la propiedad *Transaction* (*IDbTransaction*) está establecida en una transacción local iniciada por el usuario. Si

no encuentra ninguna de estas transacciones, *LINQ to SQL* inicia una transacción local (*IDbTransaction*) y la usa para ejecutar los sentencias *SQL* generadas. Cuando se han completado todas las sentencias *SQL* correctamente, *LINQ to SQL* confirma la transacción local y devuelve un valor.

4.4.2 TRANSACCIÓN LOCAL EXPLÍCITA.

En la segunda opción, si se desea administrar la transaccion de forma manual, *DataContext* proporciona la habilidad para usar una transaccion sobre la conexión que haya sido abierta por el objeto *DataContext*. En este caso se puede hacer la llamada al método *BeginTransaction* en *DataContext.Connection* antes de tratar de realizar los cambios en la base de datos. Cuando se llama a *SubmitChanges*, si la propiedad *Transaction* se establece en una transacción, la llamada a *SubmitChanges* se ejecuta en el contexto de la misma transacción. Es necesario confirmar o revertir la transacción después de su correcta ejecución, debidoa esto, después de que los cambios hayan sido realizados, se puede realizar un *commit* o un *rollback*.

Para demostrar estas dos alternativas se tiene el siguiente fragmento de código:

```
BDEjemploDataContext bd= new BDEjemploDataContext();
try
{
   //manipulando la transaccion através del objeto DataContext
   bd.Connection.Open();
   //Se inicia la transaccion, en este caso no se pueden declarar transacciones
   //de forma paralela,es decir, no se pueden declarar más de una instancia
   //de objetos DataContext
   bd.Transaction = bd.Connection.BeginTransaction();
   Libro editandoLibro = bd.Libros.Where(q => q.IDLibro == 2).Single();
   Console.Write("1 ->" + editandoLibro.Titulo);
   editandoLibro.Titulo = " BDEjemploDataContext ";
   //se escoge la opcion ConflictMode.
   bd.SubmitChanges(ConflictMode.ContinueOnConflict);
   //Si no ocurre un error se procede a realizar un commit
   bd.Transaction.Commit();
   bd.Connection.Close();
   }catch(ChangeConflictException ex){
   //en caso contrario de realiza un rollback
   bd.Transaction.Rollback();
   Console.WriteLine(ex.Message);
}
```

La desventaja en la administración de las transacciones de forma directa através de *DataContext*, es que no se pueden abarcar múltiples conexiones con *DataContext*, o mejor dicho, múltiples objetos *DataContext*.

4.4.3 TRANSACCIÓN DISTRIBUIBLE EXPLÍCITA.

Como tercera opción se encuentra el uso de las *API* de *LINQ to SQL* (por ejemplo *SubmitChanges*) dentro de un ámbito de un objeto Transaction activo. En este caso *LINQ to SQL* averigua que la llamada a *SubmiteChanges* está en el ámbito de una transacción y no crea una nueva transacción. *LINQ to SQL* también impide el cierre de la conexión en ese caso. Puede ejecutar consultas y el método *SubmitChanges* en el contexto de este tipo de transacción.

En este caso se hace uso de *TransactionScope* en el espacio de nombres *System.Transactions*. A partir de *.NET Framework 2.0* se encuentra el objeto *TransactionScope*, que permite simplificar el trabajo con transacciones. Lo primero que debe hacer es hacer una referencia a la librería *System.Transactions* en el proyecto. Este objeto distribuye (cuando se genera una secuencia de transacciones, las organiza de manera jerárquica) de forma automática la transacción sobre la base de los objetos para la cuál es usada y para el código en el que es escrito ó que envuelve, de esta manera, por ejemplo, si el ámbito (*scope*) de la aplicación abarca sólo una

única llamada a una base de datos, se utiliza una simple transacción para esa base de datos. Cuando en la transacción intervienen más de un recurso transaccional (diferentes bases de datos) la transacción deja de ser una transacción ligera para pasar a ser una transacción distribuida. Para ejemplificar esta tercera opción, considérese el siguiente fragmento de código que hace uso de *TransactionScope* y cuya explicación se verá de forma inmediata:

```
using (TransactionScope scope = new TransactionScope())
{
   try
   {
      BDEjemploDataContext bd = new BDEjemploDataContext();
      Libro editandoLibro = bd1.Libros.Where(q => q.IDLibro == 2).Single();
      Console.Write("1 ->" + editandoLibro.Titulo);
      editandoLibro.Titulo = " Nueva definicion de titulo";
      context.SubmitChanges(ConflictMode.ContinueOnConflict);
      scope.Complete();
   }catch(Exception ex){
      Console.WriteLine(ex.Message+" Error - Se deshacen los cambios");
   }
}
```

En el código, se observa que se envuelve el código en el que se accesa a la base de datos a través de un objeto DataContext (*BDEjemploDataContext*) y se modifica un libro con id=2, con un objeto *TransactionScope*, que crea a su vez una transacción controlada por un componente de *Windows*, conocido como ambito o contexto de la transacción. Posteriormente, se obtiene un objeto de la base de datos y se cambia algunos de sus datos, si todo ha salido bien, se confirma la operación de la transacción con el método *Complete()*.

La forma en que se realiza un *commit* o r*ollback* es de manera implicita y sucede de forma oculta para el programador. Cuando se crea una transacción utilizando *TransactionScope*, se crea una transacción ligera que será controlada por el componente de *Windows*, por lo que no es necesario hacer un r*ollback* de forma explícita, una vez que se sale del ámbito de la transacción (representada por la claúsula using que contiene a *TransactionScope*) los cambios se deshacen automáticamente si no han sido confirmados. Así, si se omite la llamada al metodo *Complete* despúes de realizar los cambios con el método *SubmitChanges*, no se realizará ningún cambio en la base de datos. Si una excepción es arrojada en *SubmitChanges*, el método Complete será omitido y pasado por alto, por lo que ningún cambio será efectuado, pero en el ejemplo anterior se coloco el bloque *try catch* para ejemplificar cuándo ocurre el error.

4.5 EJECUCIÓN DE CONSULTAS SQL DIRECTAMENTE HACIA LA BASE DE DATOS.

Dado que todos los métodos en una aplicación que pueden estar disponibles de forma local no pueden ser ejecutados por *SQL*, *LINQ to SQL* trata de convertirlos en operaciones y funciones equivalentes en *SQL* por lo que la mayoría de los métodos y operadores de *.NET Framework* tienen comandos equivalentes en *SQL*. Algunos de estos comandos se pueden generar a partir de las funciones que están disponibles en *LINQ to SQL*. Aquéllos que no se pueden generar, inician excepciones en tiempo de ejecución. En este caso el método *ExecuteQuey* está diseñado para para ejecutar una consulta *SQL* directamente y después convertir el resultado de la consulta directamente en objetos.Para el siguiente ejemplo se declara una consulta en *SQL* para devolver los libros contenidos en la tabla *Libro* de la base de datos *DBEjemplo*:

```
BDEjemploDataContext bd = new BDEjemploDataContext();
IEnumerable<Libro> query=bd.ExecuteQuery<Libro>("SELECT * FROM Libro");
foreach(Libro item in query) Console.WriteLine(item.Titulo);
```

Para que los resultados de una consulta *SQL* sean soportados por la aplicación y se puedan utilizar como objetos, los nombres de las columnas de los resultados deben coincidir con las

propiedades de columna de la clase de entidad. El método *ExecuteQuery* también permite el uso de parámetros. El código como el siguiente ejecuta una consulta con parametros:

```
BDEjemploDataContext bd = new BDEjemploDataContext();
IEnumerable<Libro> query=bd.ExecuteQuery<Libro>("SELECT * FROM Libro WHERE IDLibro = {0}","1");
foreach(Libro item in query) Console.WriteLine(item.Titulo);
```

4.6 CARGA APLAZADA.

Este tipo de carga un objeto solo tiene los datos que necesita. De esta forma, por ejemplo, cuando se carga una objeto Libro de la base de datos, usando este tipo de modelo, solamente los datos escenciales son cargados, es decir, la información personal de esa entidad. Posteriormente, cuando en alguna parte de la aplicación se necesita obtener los autores asociados a estos objetos, la información que no a sido cargada es localizada en la base de datos y cargada de nuevo, *LINQ to SQL* es capaz de reconocer automáticamente cualquier situación en donde los datos perdidos son requeridos. La carga aplazada es una característica del modelo de objetos (entiéndase como el mapeo de la base de datos) que representa el dominio de una aplicación que indica la habilidad que tiene la capa de acceso a datos (*DAL* por sus siglas en ingles) para cargar en una petición, a una sección de objetos que son necesitados a través de ciertas regla y cuando se agrega esta habilidad a nuestra capa de acceso a datos, se implementa el patron de carga aplazada (*Lazy Load*). La carga aplazada se encuentra habilitada de forma predeterminada en *LINQ to SQL*, para entender mejor considérese el siguiente fragmento de código:

```
BDEjemploDataContext bd = new BDEjemploDataContext();
var query = from q in bd.Autors select q;
int i = 0;
foreach (Autor item in query)  {
    Console.WriteLine(item.Nombre);
    foreach (Libro subitem in item.Libros)
           Console.WriteLine(subitem.Titulo);
}
```

En el código anterior, se obtienen los autores y por cada autor se obtienen los libros de cada uno. Al tener la carga aplazda de forma predeterminada, lo que sucede en el motor de *LINQ to SQL*, es que se ejecutan tantas consultas como números de libros tiene cada autor, es decir, si en la base de datos existen 7 libros, se ejecutarán 7 consultas *SQL* para obtener cada dato del libro que pertenece a un autor específico, además de las 7 consultas que se generan y ejecutan, se crea otra más, la que se encarga de obtener a los autores, pero esta sucede solo una vez:

```
 Aqui se obtienen los autores ----------------------------------------

SELECT [t0].[IDAutor], [t0].[Nombre], [t0].[Apellidos]
FROM [dbo].[Autor] AS [t0]

Apartír de aqui se van obteniendo los libros-----------------------------
SELECT [t0].[IDLibro], [t0].[Titulo], [t0].[IDAutor], [t0].[IDEditorial],
[t0].[FechaPublicacion], [t0].[Ejemplares], [t0].[Notas]
FROM [dbo].[Libro] AS [t0]
WHERE [t0].[IDAutor] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [1]

SELECT [t0].[IDLibro], [t0].[Titulo], [t0].[IDAutor], [t0].[IDEditorial],
[t0].[FechaPublicacion], [t0].[Ejemplares], [t0].[Notas]
FROM [dbo].[Libro] AS [t0]
WHERE [t0].[IDAutor] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [2]

SELECT [t0].[IDLibro], [t0].[Titulo], [t0].[IDAutor], [t0].[IDEditorial],
[t0].[FechaPublicacion], [t0].[Ejemplares], [t0].[Notas]
FROM [dbo].[Libro] AS [t0]
WHERE [t0].[IDAutor] = @p0
```

```
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [3]

SELECT [t0].[IDLibro], [t0].[Titulo], [t0].[IDAutor], [t0].[IDEditorial],
[t0].[FechaPublicacion], [t0].[Ejemplares], [t0].[Notas]
FROM [dbo].[Libro] AS [t0]
WHERE [t0].[IDAutor] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [5]
SELECT [t0].[IDLibro], [t0].[Titulo], [t0].[IDAutor], [t0].[IDEditorial],
[t0].[FechaPublicacion], [t0].[Ejemplares], [t0].[Notas]
FROM [dbo].[Libro] AS [t0]
WHERE [t0].[IDAutor] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [6]

SELECT [t0].[IDLibro], [t0].[Titulo], [t0].[IDAutor], [t0].[IDEditorial],
[t0].[FechaPublicacion], [t0].[Ejemplares], [t0].[Notas]
FROM [dbo].[Libro] AS [t0]
WHERE [t0].[IDAutor] = @p0
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [7]
```

Esta caracterísica puede ser útil o no en diferenes escenarios, puesto que se trata de un ajuste opcional que ofrece el comportamiento anterior descrito, en caso de no ser necesario, puede desactivarse. La carga aplazada es controlada a nivel de una entidad *DataContext*, somo se muestra a continuación:

```
BDEjemploDataContext bd = new BDEjemploDataContext();
bd.DeferredLoadingEnabled = false;
```

Si de antemano, se sabe que se va a trabajar de esta forma, obtener un objeto de la base de datos y de estos objetos obtener los que se encuentran asociados, es recomendable dejar a *LINQ to SQL* tener activada esta opción, ya que de lo contrario, los datos relacionados no aparecerán:

```
BDEjemploDataContext bd = new BDEjemploDataContext();
bd.DeferredLoadingEnabled = false;
var query = from q in bd.Autors select q;

foreach (Autor item in query)
{
     Console.WriteLine(item.Nombre);
    //A partir de aquí ninguna consulta se generará y ejecutará
    //por lo que no se mostrará resultado alguno
     foreach (Libro subitem in item.Libros)
     {
     Console.WriteLine(subitem.Titulo);
     }
}
```

CAPITULO V. DESARROLLO DE UN CASO PRÁCTICO.

5.1 INTRODUCCIÓN.

En los capítulos anteriores, se ha mostrado qué es *LINQ* y las extensiones que se basan en este. Se ha comprendido que *LINQ* no solamente es una tecnología de consulta genérica, sino también un conjunto de herramientas que pueden usarse para tratar datos relacionales, *xml*, colecciones de objetos en memoria y otros tipos de fuente gracias a la extensibilidad de *LINQ*. También se revisó *LINQ to SQL* y los conceptos que permiten al programador manipular de forma más eficaz los datos provenientes de una base de datos. En este capítulo se proyectan los conceptos vistos anteriormente sobre una implementación de una aplicación que aproveche la herramienta *LINQ to SQL*. Como parte final de la tesis, se presenta un caso práctico el cual está basado en lo que a menudo se utiliza en la vida cotidiana como es el caso de las transacciones basadas en ventas de artículos por internet.

5.2 DESCRIPCIÓN DEL PROBLEMA.

Arte-ON-LINE es una asociación que agrupa artesanos. A través de esta asociación se promueve el comercio de las artesanías pertenecientes a los artesanos. Asimismo, reúne a los artesanos para obtener productos excepcionales en diseño y calidad teniendo así un gran consorcio entre los artesanos y sus productos. Además, en su expansión busca brindar una infraestructura empresarial que facilite el acceso de dichos productos a los clientes, esto se pretende realizar a través de un sistema para que se registren y puedan acceder a un producto disponible de los artesanos.

5.3 REQUERIMIENTOS.

Las funciones que son requeridas y que deben ser implementadas para poder satisfacer las necesidades tanto de la asociación como de los clientes son las siguientes:

1. Administración de la base de datos para:
   - Artesanos.
     Dar de alta. Mantener actualizada la información. Posibilidad de eliminar artesanos.
   - Artesanías.
     Dar de alta. Mantener actualizada la información. Posibilidad de eliminar artesanías.
   - Las órdenes de los clientes.
     Administración de las órdenes creadas provenientes de la creación del carrito de compra. Poder observar las órdenes que realizaron los clientes a través de Internet. Modificar el estado en el que se encuentra una orden, ya que esto permitirá realizar un seguimiento de las artesanías para que lleguen al cliente.

2. Acceso de los usuarios a través de Internet para las artesanías con el fin de:
   - Realizar el proceso de registro y login para los clientes de Internet. Seguimiento de las órdenes por parte de los clientes. Observar y realizar búsquedas de artesanías que se encuentran en la base de datos.

Para poder cumplir con las expectativas de la tesis y cubrir los elementos más importantes de *LINQ to SQL*, se requiere de 2 aplicaciones que operarán en dominios diferentes. La primera servirá para administrar la base de datos contenida en *SQL Server 2005 Express Edition* y será a la que tendrán acceso para realizar operaciones (inserción, actualización y eliminación) sobre los datos. La segunda se enfocará hacia la venta de las artesanías con los clientes provenientes de la Web, incluyendo el registro de estos últimos. Todas las aplicaciones se desarrollarán en el sistema

operativo Windows Vista Ultimate, además de que la aplicación ASP.NET será publicada en el servidor IIS (Internet Information Services) 7.0, el cual viene incluido en este sistema.

5.4 ANÁLISIS Y DISEÑO.

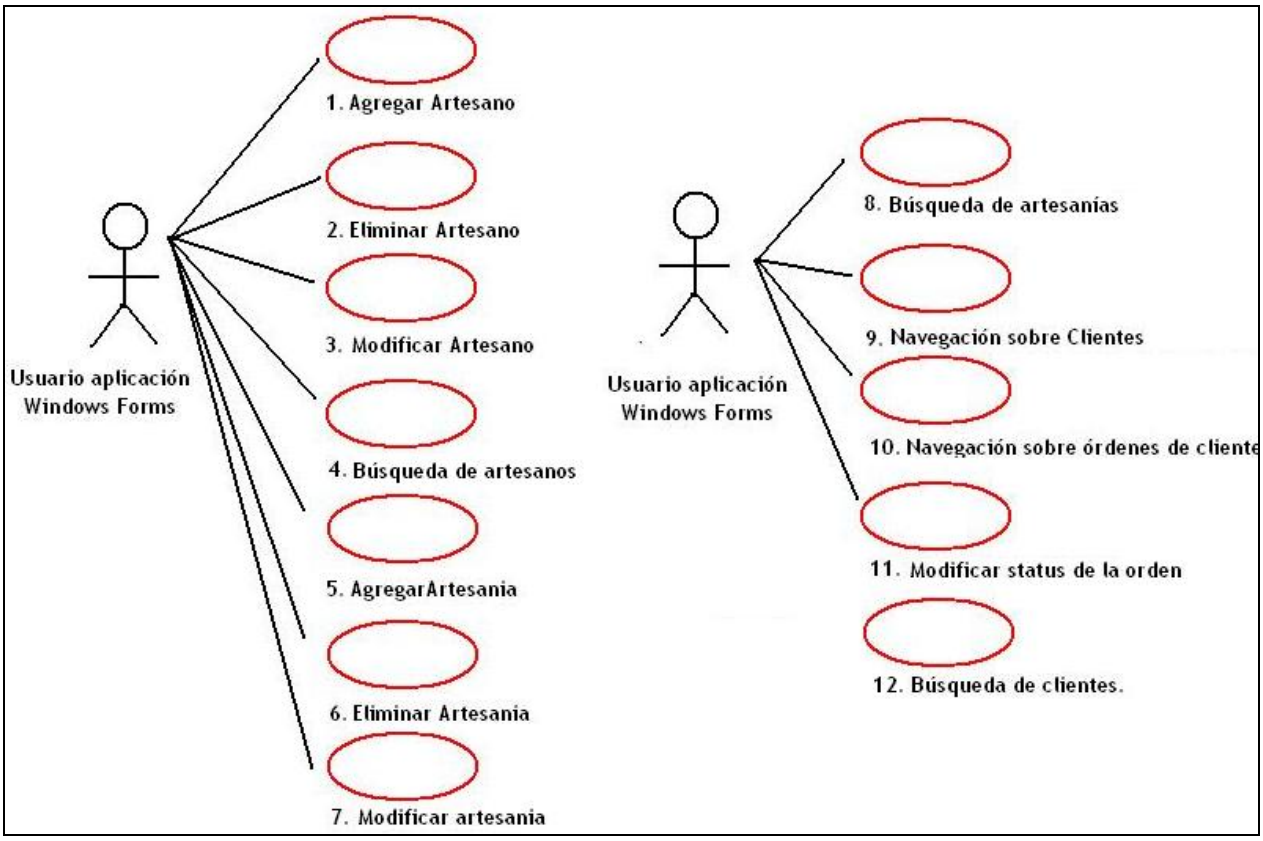
El uso de *LINQ* se ve reflejado en el desarrollo de la arquitectura de la aplicación ya que resulta un factor muy importante para poder tener un mejor control tanto en tiempo, como en herramientas que traduzcan todo el esfuerzo en una aplicación mejorada.

5.4.1 DISEÑO DE CASOS DE USO.

En la siguiente imagen se tienen los casos de uso que se diseñaron para poder cumplir con las expectativas de funcionamiento tanto para la aplicación Windows Forms que servirá para la administración de la base de datos, como para la aplicación Web en ASP.NET para el caso de la página Web de ArteOnLine:

5.4.1.1 Aplicación Windows Forms.

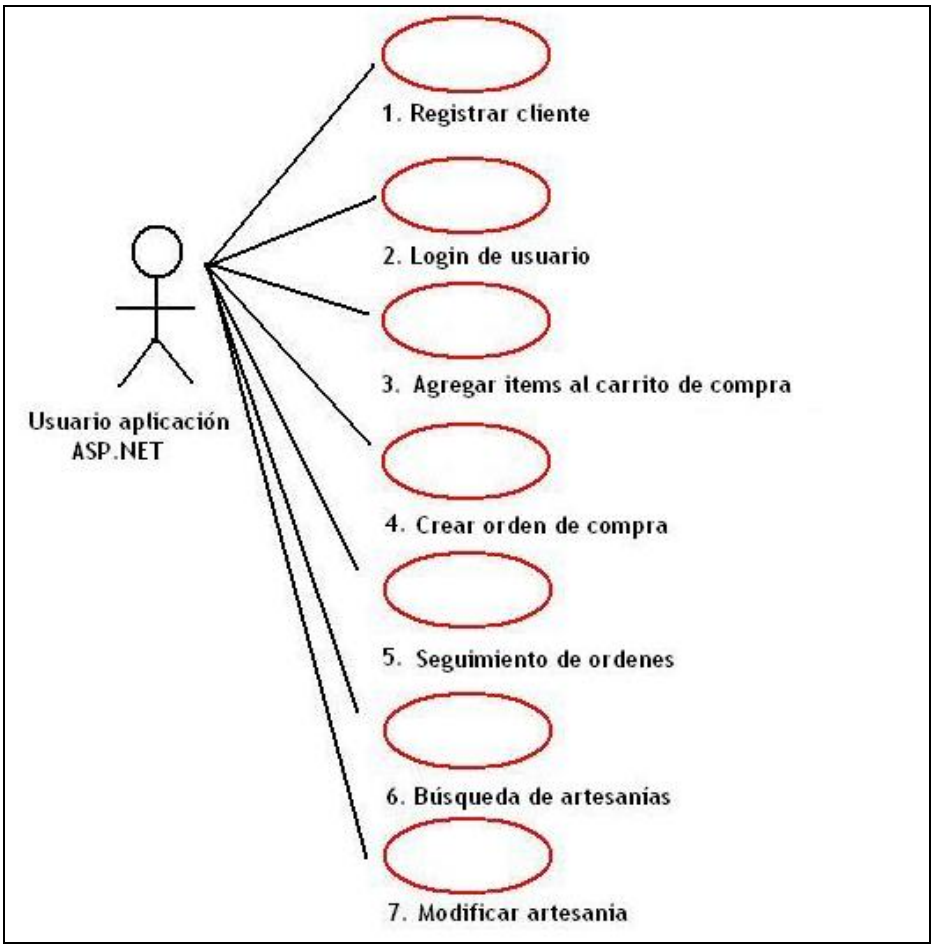
La definición de los casos de uso para esta aplicación se muestra en la siguiente imagen:



Casos de uso aplicación Windows Forms.

5.4.1.2 Aplicación ASP.NET.

La definición de los casos de uso para esta aplicación se muestra en la siguiente imagen:



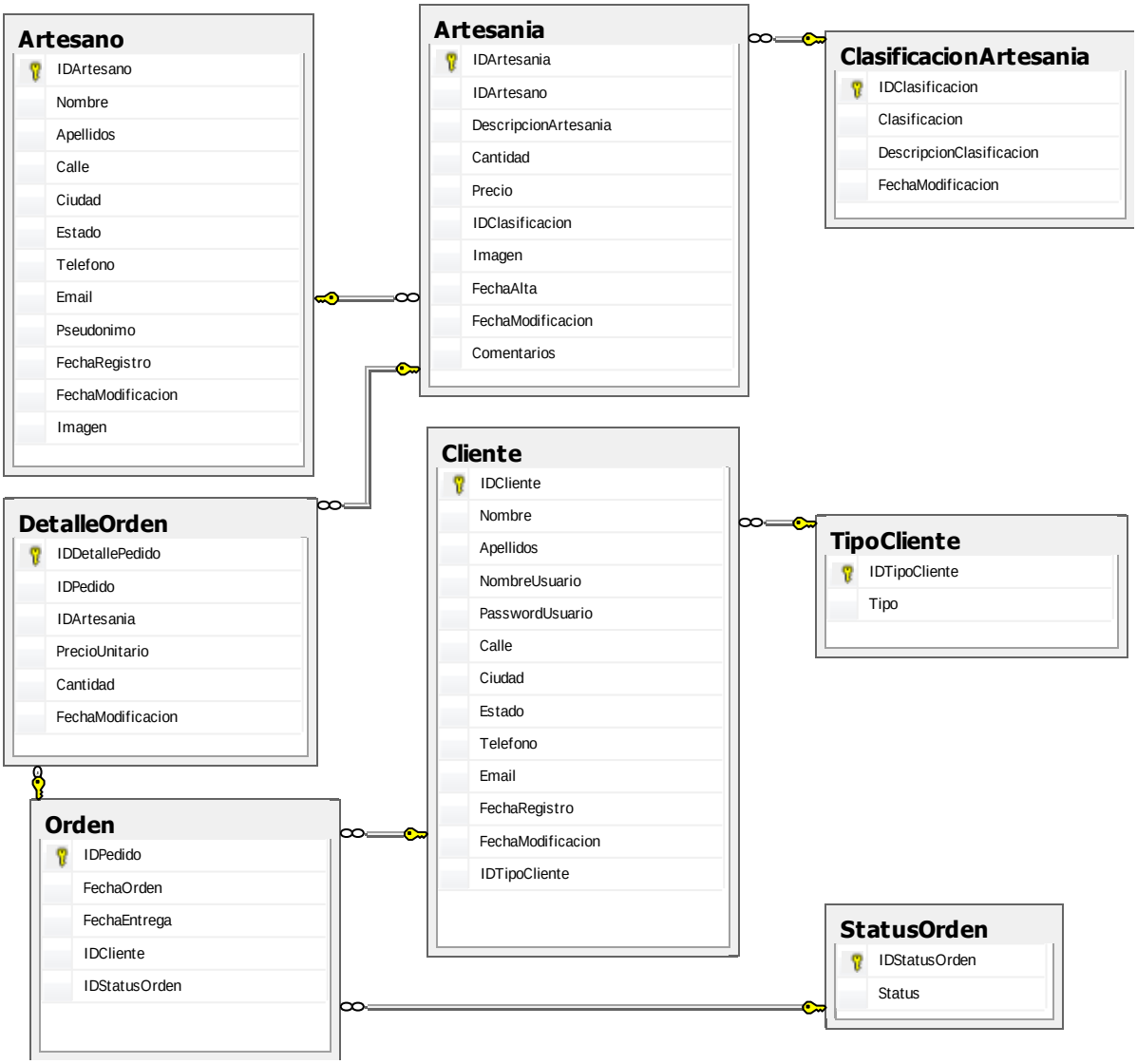
Casos de uso aplicación ASP.NET

Nota: Para poder observar a detalle cada caso de uso favor de referirse al Anexo A de esta tesis.

5.4.2 DISEÑO DE LA BASE DE DATOS.

La base de datos de ArteOnLine estará albergada en *SQL Server Express Edition 2005*. A continuación se muestra al diagrama Entidad-Relación correspondiente al esquema de definición haciendo uso del lenguaje de definición de datos (*DML*) para *SQL Server*:

Nota: Para observar el lenguaje SQL usado favor de referirse al Anexo B de esta tesis.

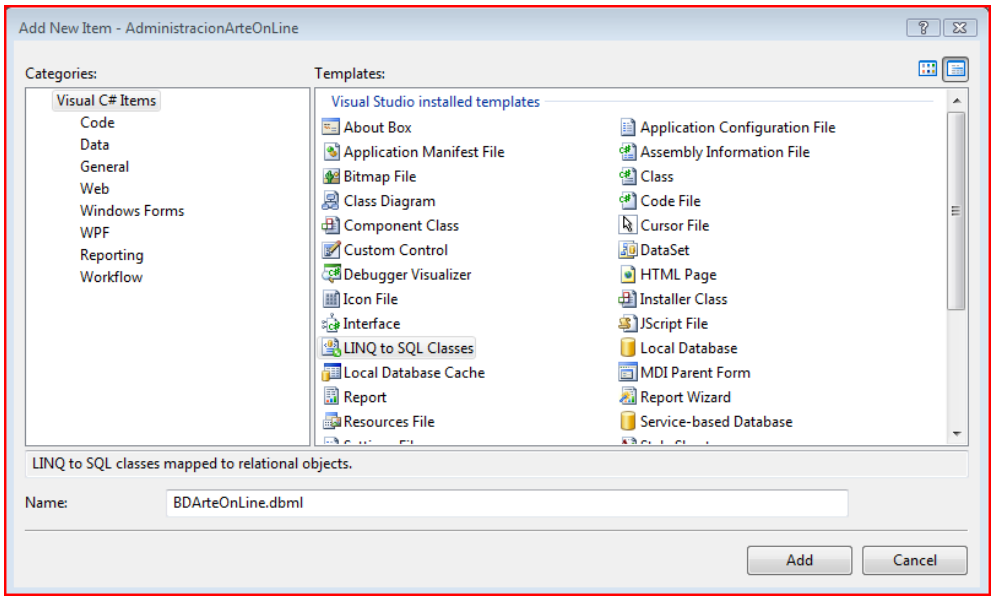


5.5 DESARROLLO E IMPLEMENTACIÓN.

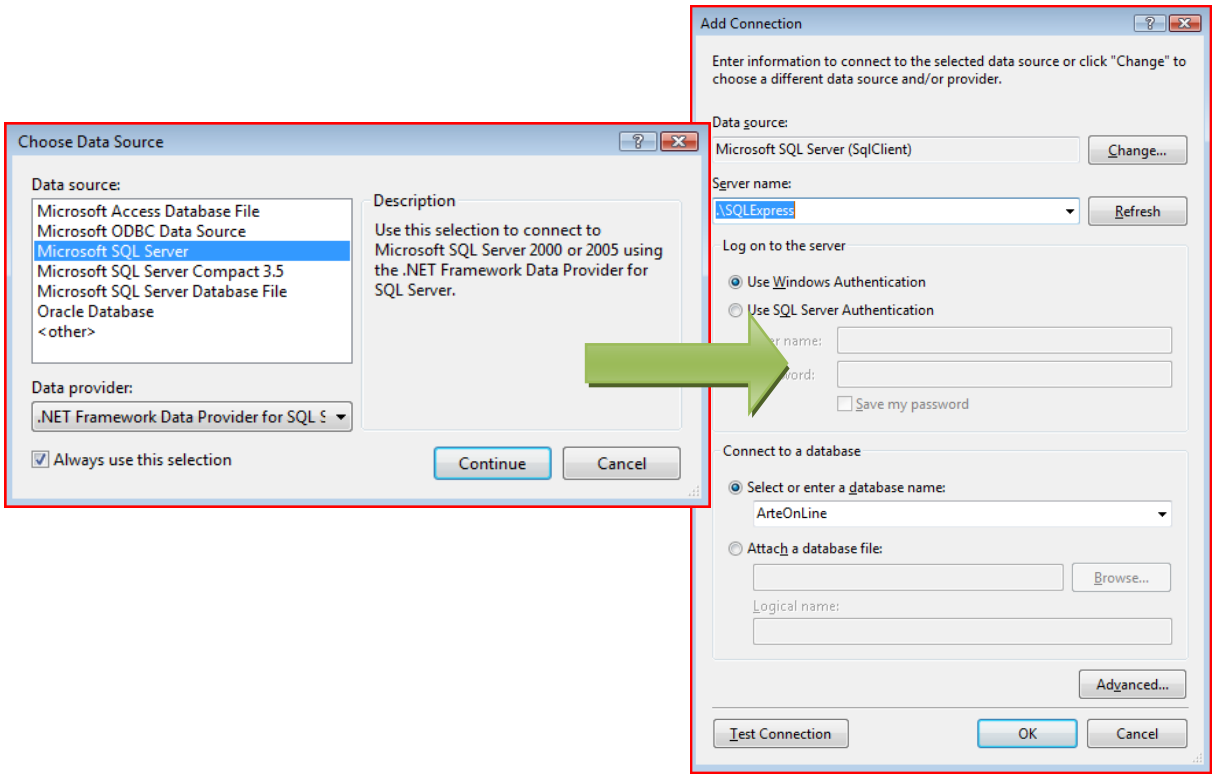
5.5.1 MAPEO DE LA BASE DE DATOS.

En primera instancia debe realizarse el mapeo de la base de datos definida anteriormente (*ArteOnLine*) contenida en *SQL Server 2005*, para poder trabajar sobre ella con *LINQ to SQL* y poder implementar el comportamiento de la lógica de negocio. Con el proceso de mapeo se generará el modelo de objetos y el archivo de lenguaje intermedio de marcado de base de datos (*.dbml*). Para poder realizar el mapeo se seguirán los pasos descritos en el capítulo 3 en la sección 3.1.4 a través de la interfaz gráfica del diseñador de objetos relacionales:

Desde el menú *Project* en la opción *Add New Item…* se agrega un nuevo elemento *LINQ to SQL Classes* a la solución del proyecto. Posteriormente se abre una ventana y se establece el nombre *ArteOnLine.dbml* y para finalizar dar clic sobre el botón *Add*.

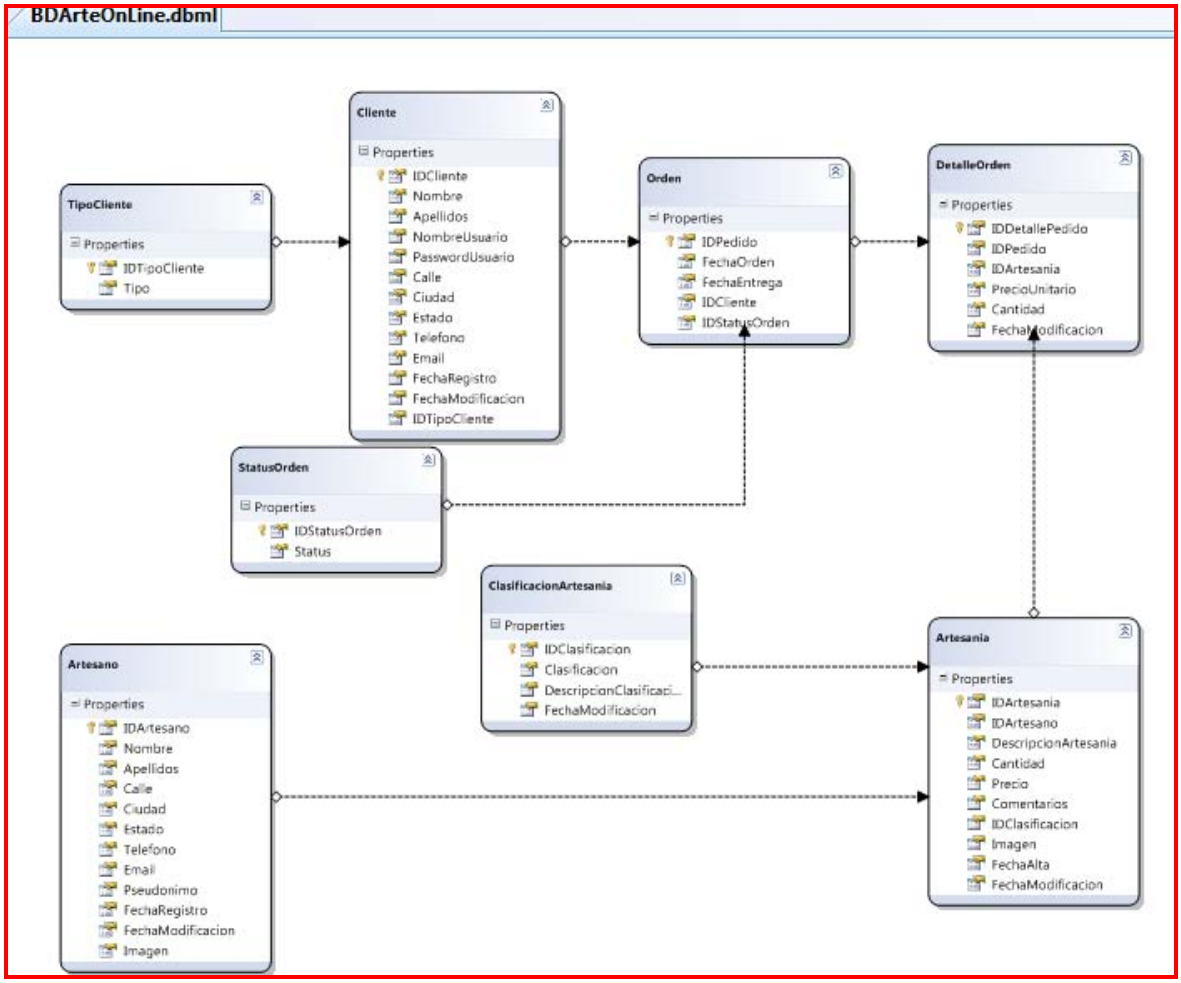


A través de la herramienta Server Explorer en *Visual Studio 2008*, conectar con la base de datos ubicada en *SQL Server 2005* proporcionando los datos de *Data Source* (*Microsoft SQL Server -> SQLClient*), *Server Name* (instancia de *SQL Server* donde se localiza la base de datos) y *Log on* (*Windows Authentication*) y en *Connect to database* seleccionar o poner el nombre de la base de datos, en este caso es *ArteOnLine*.



Una vez conectados con la base de datos, aparecerá su contenido (tablas) y las áreas del diseñador mencionadas en el capítulo 3, seleccionar todas las tablas y arrastrarlas sobre la

superficie izquierda del área de diseño. Al terminar, el proceso de mapeo tendrá la siguiente apariencia:



5.5.2 DISEÑO DEL SISTEMA PARA ADMINISTRAR LA BASE DE DATOS.

Para poder administrar la base de datos, se siguió el criterio de trabajar sobre una ventana principal en la que posteriormente y de acuerdo a un menú, se mostrarán las interfaces que operan sobre las diversas tablas que existen en la base de datos. Por razones de espacio, el código utilizado para generar esta y las demás interfaces serán omitidos, además de otros que nada tienen que ver con el manejo de la base de datos, con lo que se centrará en el código implementado para la lógica de negocio adyacente a la interfaz. Este formulario principal tiene la característica de ser un formulario *MDIParent* (*Multiple Document Interface*), lo que significa que puede contener formularios secundarios que al mostrarse, se centraran sobre el área de este formulario.

5.5.2.1 Interfaz principal PrincipalMDIParent.

En este caso el código que asociado a la carga de las interfaces para los artesanos, artesanías, ordenes y clientes son los siguientes:

```
public partial class PrincipalMDIParent : Form {
//…
Definición de métodos asociados a la interfaz
//…
//Definición de métodos para la carga de las demás interfaces
```

```
private void artesaniasToolStripMenuItem_Click(object sender, EventArgs e) {
    GUIArtesanias artesaniasForms = new GUIArtesanias();
    this.StripStatusLabel.Text = "Cargando ventana artesanías...";
    Application.DoEvents();
    artesaniasForms.MdiParent=this;
    artesaniasForms.WindowState = FormWindowState.Normal;
    artesaniasForms.Location = new Point(0, 0);
    artesaniasForms.StartPosition = FormStartPosition.Manual;
    artesaniasForms.Show();
    this.StripStatusLabel.Text = "Ventana artesanías cargada";
}

private void ArtesanosToolStripMenuItem_Click(object sender, EventArgs e) {
    GUIArtesanos artesanosForm = new GUIArtesanos();
    this.StripStatusLabel.Text = "Cargando ventana artesanos...";
    Application.DoEvents();
    artesanosForm.MdiParent = this;
    artesanosForm.WindowState = FormWindowState.Normal;
    artesanosForm.Location = new Point(0, 0);
    artesanosForm.StartPosition = FormStartPosition.Manual;
    artesanosForm.Show();
    this.StripStatusLabel.Text = "Ventana artesanos cargada";
}

private void clienteToolStripMenuItem Click(object sender, EventArgs e) {
   //Código asociado para la carga de interfaz GUIClientes
}

private void ordenesToolStripMenuItem Click(object sender, EventArgs e) {
   //Código asociado para la carga de interfaz GUIOrdenes
}
}
```

El primer método, *ArtesaniasToolStripMenuItem_Click*, el cual está asociado al evento clic del elemento *ArtesaniasToolStripMenuItem* contenido en el menú de la interfaz principal, se encarga de cargar y mostrar la ventana de artesanos para que el usuario del sistema observe los datos contenidos en la base de datos.

Como se observa en los métodos presentados en el código anterior, el patrón de funcionamiento es el mismo, la única diferencia es la interfaz a la cual están asociados, de esta forma, el método ArtesanosToolStripMenuItem_*Click* está asociado a la carga de la interfaz de las artesanías, el método C*lienteToolStripMenuItem_Click* está asociado a la interfaz de los clientes y el método OrdenesToolStripMenuItem_*Click* asociado a la interfaz de las órdenes realizadas por el sistema de internet, el cual se verá más adelante. Por lo que el funcionamiento descrito para el primer método es aplicable también a estos métodos.

5.5.2.2 Artesanos.

Para trabajar con los artesanos provenientes de la base de datos, se operará sobre 3 diferentes interfaces, la cuáles son: *GUIArtesanos*, *GUIArtesanosModificaciones*, *GUIBuscarArtesanos*. La primera, es la encargada de cargar de la base de datos a los artesanos y mostrar los datos sobre una lista, por otro lado, *GUIArtesanosModificaciones* se encarga de navegar sobre los artesanos para poderlos observar a detalle y además de las operaciones de inserción y modificación de los registros. La última interfaz, denominada *GUIArtesanosBuscar* simplemente realiza una búsqueda sobre la tabla artesano de acuerdo a su nombre y a un valor dado por el usuario para el mismo nombre.

5.5.2.2.1 Interfaz GUIArtesanos.

Los métodos que servirán para cualquier operación se enlistan a continuación:

5.5.2.2.1.1 *Método InicializarArtesanosListView*.

Este método prepara al control *ListView* para recibir y mostrar los datos, además de que se le agregan columnas para los datos o campos de los artesanos para mostrarlos al usuario. El código asociado a esta función es el siguiente.

```
private void InicializarArtesanosListView(){
    this.ArtesanosListView.Columns.Add("Nombre", 100);
    this.ArtesanosListView.Columns.Add("Apellidos", 150);
    this.ArtesanosListView.Columns.Add("Calle", 100);
    this.ArtesanosListView.Columns.Add("Ciudad", 100);
    this.ArtesanosListView.Columns.Add("Estado", 100);
    this.ArtesanosListView.Columns.Add("Telefono", 150);
    this.ArtesanosListView.Columns.Add("Email", 100);
    this.ArtesanosListView.Columns.Add("Pseudonimo", 100);
    this.ArtesanosListView.Columns.Add("Fecha Registro", 100);
    this.ArtesanosListView.Columns.Add("Fecha Modificacion", 100);
    this.ArtesanosListView.View = View.Details;
    this.ArtesanosListView.FullRowSelect = true;
}
```

5.5.2.2.1.2 *Método TodosToolStripMenuItem_Click*.

Este método tiene definida la conexión hacia la base de datos a través de un objeto *DataContext* asociado a la base de datos para extraer los datos de los artesanos y colocarlos sobre una lista genérica del tipo *ArtesanosLazy* para incorporarlos al *ListView* a través de una llamada al método *LlenarArtesanosListView*.

```
BDArteOnLineDataContext cargarDatosBD = new BDArteOnLineDataContext();
//Forzar la ejecución inmediata con .ToLis();
List<ArtesanosLazy> query = (from artesano in cargarDatosBD.Artesanos
                             select new ArtesanosLazy {
                                 IDArtesano = artesano.IDArtesano,
                                 Nombre = artesano.Nombre,
                                 Apellidos = artesano.Apellidos,
                                 Calle = artesano.Calle,
                                 Ciudad = artesano.Ciudad,
                                 Estado = artesano.Estado,
                                 Telefono = artesano.Telefono,
                                 Email = artesano.Email,
                                 Pseudonimo = artesano.Pseudonimo,
                                 FechaRegistro = artesano.FechaRegistro,
                                 FechaModificacion = artesano.FechaModificacion,
                                 Imagen = artesano.Imagen
                             }).ToList();
//GetRange crea una copia de un rango específico de valores y se
//le pasa hacia la lista y pasarlos al formulario para modificarlos.
listaArtesanos = query.GetRange(0, query.Count);
cargarDatosBD.Dispose();
this.LlenarArtesanosListView();
```

La clase *ArtesanosLazy* tiene los siguientes campos: *IDArtesano (int), Apellidos (string), Calle (string), Ciudad (string), Estado (string), Teléfono (string), Email (string), Pseudónimo* (string) *Fecha Registro (DateTime), Fecha Modificación (DateTime?) e imagen (Binary).*

5.5.2.2.1.3 *Método LlenarArtesanosListView*.

Con este método los resultados son visibles al usuario y se pueden realizar operaciones como modificación o eliminación.

```
private void LlenarArtesanosListView(){
    int indice = 0;
    this.ArtesanosListView.Items.Clear();
    this.ArtesanosListView.BeginUpdate();
    foreach (ArtesanosLazy artesano in listaArtesanos) {
        this.ArtesanosListView.Items.Add(artesano.Nombre);
        this.ArtesanosListView.Items[indice].SubItems.Add(artesano.Apellidos);
```

```
            this.ArtesanosListView.Items[indice].SubItems.Add(artesano.Calle);
            this.ArtesanosListView.Items[indice].SubItems.Add(artesano.Ciudad);
            this.ArtesanosListView.Items[indice].SubItems.Add(artesano.Estado);
            this.ArtesanosListView.Items[indice].SubItems.Add(artesano.Telefono);
            this.ArtesanosListView.Items[indice].SubItems.Add(artesano.Email);
            this.ArtesanosListView.Items[indice].SubItems.Add(artesano.Pseudonimo);
            this.ArtesanosListView.Items[indice].SubItems.Add(
                            artesano.FechaRegistro.ToLongDateString());
            if (artesano.FechaModificacion == null) {
                this.ArtesanosListView.Items[indice].SubItems.Add("");
            }else {
                this.ArtesanosListView.Items[indice].SubItems.Add(
                                artesano.FechaModificacion.Value.ToLongDateString());
            }
            indice++;
        }
        this.ArtesanosListView.EndUpdate();}
```

5.5.2.2.2 Interfaz GUIArtesanosModificaciones.

Los métodos que son usados de forma general son los siguientes:

5.5.2.2.2.1 *Método MostrarDatosListaArtesano*.

Este método tiene como función cargar los datos y la imagen que se encuentran en los registros de la lista genérica (provenientes de la base de datos) y pasarlos hacia cada uno de los campos correspondientes de la interfaz *GUIArtesanosModificaciones*.

```
private void MostrarDatosListaArtesano()
{
    CambioImagen = false;
    int indice = 0;
    indice = this.indice;
    this.CantidadToolStripTexBox.Text = this.cantidad.ToString();
    this.IndiceToolStripTextBox.Text = (indice + 1).ToString();
    this.IDArtesanoTextBox.Text = listaArtesanos[this.indice].IDArtesano.ToString();
    this.NombreTextBox.Text = listaArtesanos[this.indice].Nombre;
    this.ApellidosTextBox.Text = listaArtesanos[this.indice].Apellidos;
    this.CalleTextBox.Text = listaArtesanos[this.indice].Calle;
    this.CiudadTextBox.Text = listaArtesanos[this.indice].Ciudad;
    this.EstadoTextBox.Text = listaArtesanos[this.indice].Estado;
    this.TelefonoTextBox.Text = listaArtesanos[this.indice].Telefono;
    this.EmailTextBox.Text = listaArtesanos[this.indice].Email;
    this.PseudonimoTextBox.Text = listaArtesanos[this.indice].Pseudonimo;
    this.FechaRegistroTextBox.Text =
                listaArtesanos[this.indice].FechaRegistro.ToLongDateString();
    if (listaArtesanos[this.indice].FechaModificacion == null) {
        this.FechaModificacionTextBox1.Text = "";
    } else {
        this.FechaModificacionTextBox1.Text =
              listaArtesanos[this.indice].FechaModificacion.Value.ToLongDateString();
    }
    if (!(listaArtesanos[this.indice].Imagen == null)) {
        this.ImagenPictureBox.Image =
               Conversiones.BytesAImagen(listaArtesanos[this.indice].Imagen.ToArray());
    }
}
```

5.5.2.2.2.2 *Método CargarImagenButton_Click*.

Este método asociado al evento clic del botón *CargarImagenButton*, opera sobre el cambio de una imagen a partir de un archivo de imagen válido, el cuál será seleccionado a partir de un cuadro de diálogo de selección de archivos, este método es usado cuando se requiere en una operación de inserción de un registro u opcional en caso de una modificación de registro. Dentro de este método se usa un método estático de la clase *Conversiones*, *RedimensionarImagen*. Este último método es el encargado de pasar una imagen de un tamaño específico, hacia un tamaño de 100 pixeles x 100 pixeles, para ser guardada en la base de datos.

```
OpenFileDialog openFileDialog = new OpenFileDialog();
openFileDialog.InitialDirectory = rutaImagenes;
openFileDialog.Filter = "Archivos de imagenes (*.jpg)|*.jpg|Todos los archivos (*.*)|*.*";
string NombreArchivo = null;
if (openFileDialog.ShowDialog(this) == DialogResult.OK)
{
    NombreArchivo = openFileDialog.FileName;
    CambioImagen = true;
    Image imagen = Conversiones.RedimensionarImagen
                              (Image.FromFile(NombreArchivo), 100, 100, ImageFormat.Jpeg);
    this.ImagenPictureBox.Image = imagen;
}
```

5.5.2.2.3 Inserción de artesanos.

El método encargado de esta función en esta interfaz es *AgregarToolStripMenuItem_Click*, este método tiene asignada la función de cargar la interfaz GUIArtesanosModifiaciones con los campos vacios y a través de esta interfaz se van a insertar nuevos artesanos hacia la base de datos. Si en el formulario encargado de la inserción se presiona el botón indicando que las inserciones ya se realizaron, la lista que tiene los nuevos datos son asignados de nueva cuenta a la lista de este formulario y se muestran los datos, con esto se asegura que la lista mostrada estará actualizada con los nuevos artesanos previamente insertados en la base de datos. El código asociado es el siguiente:

```
//se realiza una instancia de GUIArtesanosModificaciones, se pasan el tipo
//de operacion (1 = agregar)
GUIArtesanosModificaciones dialogBox = new GUIArtesanosModificaciones(1);
dialogBox.Text = "Inserción de artesanos";
//a la lista de esta instancia se le pasan los datos de los artesanos
//(ListaArtesanos)
dialogBox.ListaArtesanos = this.listaArtesanos;
if (dialogBox.ShowDialog(this) == DialogResult.OK) {
    //la lista listaArtesanos del formulario GUIArtesanosModificaciones
    //a la que se le agregaron nuevos datos del tipo ArtesanosJoin)
    //se asigna a la lista de este formulario para tener los datos actualizados
    this.listaArtesanos = dialogBox.ListaArtesanos;
    //se muestran los datos
    this.LlenarArtesanosListView();
}
dialogBox.Dispose();
```

Por su parte la interfaz *GUIArtesanosModificaciones* para esta operación tiene los siguientes métodos:

5.5.2.2.3.1 *Método EjecutarButton_Click*.

Este método asociado al evento clic del botón *EjecutarButton* crea un nuevo objeto del tipo *ArtesanosLazy* para insertarlo en la lista y un objeto Artesano para la base de datos y se realiza la inserción. Además, maneja dos excepciones, la primera de ellas está relacionada con una excepción lanzada por el sistema manejador de base de datos cuando se actualiza algún registro, la segunda excepción está relacionada con el tipo de datos esperados en algunos de los campos.

```
using (TransactionScope scope = new TransactionScope())
{
  try
  {
     //se obtienen los datos de los controles a las siguientes variables
     // string nombreArtesanoNuevo ,string apellidosArtesanoNuevo
     // string calleArtesanoNuevo ,string ciudadArtesanoNuevo
     // string estadoArtesanoNuevo , string telefonoArtesanoNuevo
     // string emailArtesanoNuevo, string pseudonimoArtesanonuevo
     // Binary imagenNueva
```

```
     if (nombreArtesanoNuevo == null || calleArtesanoNuevo == null ||
         ciudadArtesanoNuevo == null || estadoArtesanoNuevo==null)
     {
        //Mostrar Cuadro de mensaje de campos vacios
        return;
      }

      //Si se va a Insertar
      if (operacion == 1) {
         if (CambioImagen) { imagenNueva =
Conversiones.ImagenABytes(this.ImagenPictureBox.Image, ImageFormat.Jpeg); }
         else { imagenNueva =
Conversiones.ImagenABytes(Conversiones.RedimensionarImagen(Image.FromFile(rutaImagenes +
@"\Imagen no Disponible.jpg"), 100, 100, ImageFormat.Jpeg), ImageFormat.Jpeg); }
         //Instanciar para agregarlo a la lista
         ArtesanosLazy nuevoArtesanoLista = new ArtesanosLazy()
         {
            //a las propiedades se le asignan valores para actualizar la lista
            Nombre = nombreArtesanoNuevo,
            Apellidos = apellidosArtesanoNuevo,
            Calle = calleArtesanoNuevo,
            Ciudad = ciudadArtesanoNuevo,
            Estado = estadoArtesanoNuevo,
            Telefono = telefonoArtesanoNuevo,
            Email = emailArtesanoNuevo,
            Pseudonimo = pseudonimoArtesanonuevo,
            Imagen = imagenNueva,
            FechaRegistro = this.FechaActual
         };
         //Instanciar para agregarlo a la BD
         Artesano artesanoNuevoBD = new Artesano {
            Nombre = nombreArtesanoNuevo,
            Apellidos = apellidosArtesanoNuevo,
            Calle = calleArtesanoNuevo,
            Ciudad = ciudadArtesanoNuevo,
            Estado = estadoArtesanoNuevo,
            Telefono = telefonoArtesanoNuevo,
            Email = emailArtesanoNuevo,
            Pseudonimo = pseudonimoArtesanonuevo,
            Imagen = imagenNueva,
            FechaRegistro = this.FechaActual
         };
         //se inserta la lista para agregarla a la BD
         this.listaArtesanosInsertar.Add(artesanoNuevoBD);
         //Se agrega el artesano ala lista para actualizarla
         this.listaArtesanos.Add(nuevoArtesanoLista);

         this.cantidad++;
         this.IndiceToolStripTextBox.Text = cantidad.ToString();
         this.CantidadToolStripTexBox.Text = cantidad.ToString();
         foreach (Control c in this.Controls) {
            if (c is TextBox) { c.Text = ""; }
            if (c is PictureBox) { PictureBox m = (PictureBox)c; m.Image = null; }
         }
         this.FechaRegistroTextBox.Text = this.FechaActual.ToLongDateString();

      } else if (operacion == 2) {
         //Código asociado a una modificación
  } catch (SqlException) {
      //Mostrar Cuadro de mensaje de error
  } catch (FormatException) {
      //Mostrar Cuadro de mensaje de datos incorrectos o campos vacíos
  }
}
```

5.5.2.2.3.2 *Método OKButton_Click*.

Este método asociado al evento clic del botón *OKButton* trabaja sobre la operación de inserción de datos. Además de que se asigna a la propiedad *DialogResult* de esta interfaz a un valor *Yes*, por lo que se puede considerar que este valor es una confirmación de que los

procedimientos de inserción fueron correctos, caso contrario cuando sucede algún error, en la que se devuelve un valor *No*. Esto nos permite saber que en esta interfaz se terminaron de realizar las operaciones necesarias y de esta forma efectuar la actualización tanto de la lista como del *ListView* de la interfaz *GUIArtesanos*. En este método se efectúa lo siguiente:

```
//si es insertar
using (TransactionScope scope = new TransactionScope()){
    try {
        if (operacion == 1) {
            int registros = this.listaArtesanosInsertar.Count;
            if (registros > 0) {
                BDArteOnLineDataContext instanciaInsertar = new BDArteOnLineDataContext();
                //Agregar los artesanos a la BD a través del método sp_CrearArtesano
                // y este nos regresará el valor del identity creado para asignárselo
                //al elementos de la lista listaArtesanos y actualizar el IDArtesano
                //de esa lista
                for (int i = 0; i < registros; i++) {
                    this.listaArtesanos[this.elementosListaArtesanos].IDArtesano =
                            instanciaInsertar.sp_CrearArtesano(
                            listaArtesanosInsertar[i].Nombre,
                            listaArtesanosInsertar[i].Apellidos,
                            listaArtesanosInsertar[i].Calle,
                            listaArtesanosInsertar[i].Ciudad,
                            listaArtesanosInsertar[i].Estado,
                            listaArtesanosInsertar[i].Telefono,
                            listaArtesanosInsertar[i].Email,
                            listaArtesanosInsertar[i].Pseudonimo,
                            listaArtesanosInsertar[i].FechaRegistro,
                            listaArtesanosInsertar[i].Imagen);
                    this.elementosListaArtesanos++;
                }
                this.listaArtesanosInsertar.Clear();
                scope.Complete();
                instanciaInsertar.Dispose();
                scope.Dispose();
                MessageBox.Show("Los artesanos fueron agregados existosamente",
                                "Inserción",MessageBoxButtons.OK, MessageBoxIcon.Information);
            }
            this.DialogResult = DialogResult.OK;
        }
    } catch (SqlException) {
        this.DialogResult = DialogResult.Abort;
        //Mostrar Cuadro de mensaje de error
    }
}
```

Como se observa se hace uso del método *sp_CrearArtesano*, que representa a un procedimiento almacenado. Este último método sirve para poder agregar los artesanos que se encuentran en la lista genérica *listaArtesanosInsertar* uno por uno y devolver al mismo tiempo el id del artesano recién agregado, con lo que al mismo tiempo, este id del artesano recién agregado es asignado a la lista genérica *listaArtesanos* para ser devuelta, cuando sea requerida, a la interfaz *GUIArtesanos* y de esta forma tener todos los datos actualizados.

Como se mencionó anteriormente, se hace uso de un procedimiento almacenado el cual está definido en la base de datos *ArteOnline* dentro de *SQL Server 2005*, esté procedimiento almacenado se encuentra definido de la siguiente forma:

```
create procedure sp_CrearArtesano(
       @nombre nvarchar(50),@apellidos nvarchar(50),
       @calle nvarchar(50), @ciudad nvarchar(30),
       @estado nvarchar(20), @telefono nvarchar(30),
       @email nvarchar(50), @pseudonimo nvarchar(30),
       @fechaRegistro DateTime,@imagen VarBinary(Max)
)
AS
BEGIN
DECLARE @IdArtesano int;
BEGIN TRANSACTION Artesano
```

```
INSERT INTO Artesano(Nombre, Apellidos, Calle, Ciudad, Estado, Telefono,
                     Email,Pseudonimo,FechaRegistro,Imagen)
VALUES(@nombre, @apellidos, @calle, @ciudad, @estado, @telefono, @email,
        @pseudonimo,@fechaRegistro, @imagen);

        IF(SELECT @@Error)!=0
                Begin
                        RollBack Transaction Artesano;
                        Set @IdArtesano=0;
                End
        ELSE
                Begin
                        Commit Transaction Artesano;
                        Set @IdArtesano=@@Identity;
        END
return @IdArtesano;
End
```

Como se puede observar, este procedimiento recibe como parámetros todos los datos que son requeridos para la inserción de un artesano (*nombre, apellidos, calle, ciudad, estado, teléfono, email, pseudónimo, fecharegistro* e *imagen*), una vez que se obtienen los datos se procede a insertar al artesano en la base de datos usando para ello transacciones, y a través de la sentencia *set @IdArtesano=@@Identity* se obtiene el valor del *Identity* que tiene asignado el campo *IDArtesano* de la tabla *Artesano* para finalmente ser devuelto por el procedimiento almacenado. En el siguiente fragmento de código, se muestra el resultado del mapeo del anterior procedimiento almacenado hacia un método de C# 3.0 en .NET:

```
[Function(Name="dbo.sp_CrearArtesano")]
public int sp_CrearArtesano( [Parameter(DbType="NVarChar(50)")] string nombre,
       [Parameter(DbType="NVarChar(50)")] string apellidos,
       [Parameter(DbType="NVarChar(50)")] string calle,
       [Parameter(DbType="NVarChar(30)")] string ciudad,
       [Parameter(DbType="NVarChar(20)")] string estado,
       [Parameter(DbType="NVarChar(30)")] string telefono,
       [Parameter(DbType="NVarChar(50)")] string email,
       [Parameter(DbType="NVarChar(30)")] string pseudonimo,
       [Parameter(DbType="DateTime")]     System.Nullable<System.DateTime> fechaRegistro,
       [Parameter(DbType="VarBinary(MAX)")]   System.Data.Linq.Binary imagen)
{
            IExecuteResult result = this.ExecuteMethodCall( this,
                                        ((MethodInfo)(MethodInfo.GetCurrentMethod())),
                                        nombre, apellidos, calle,
                                        ciudad, estado, telefono,
                                        email, pseudonimo, fechaRegistro, imagen);
             return ((int)(result.ReturnValue));
}
```

5.5.2.2.4 Modificación de artesanos.

El método encargado de realizar esta operación en la interfaz *GUIArtesanos* es el método *ModificarToolStripMenuItem_Click*, el cual carga la interfaz *GUIArtesanosModifiaciones* con los datos que se tienen en la lista que se muestra y a través de la interfaz *GUIArtesanosModifiaciones* se van a mostrar los datos en campos donde se pueden modificar los valores de los campos que tenga en la base de datos, para posteriormente efectuar los cambios sobre la base de datos. Lo que hará también el botón de la interfaz *GUIArtesanosModificaciónes* es modificar la propiedad *DialogResult* de esta misma interfaz a un valor *OK* y posteriormente se cierre, por lo que la lista que tiene los datos modificados son asignados de nueva cuenta a la interfaz *GUIArtesanos*. El código que efectúa lo mencionado es el siguiente:

```
if (this.ArtesanosListView.Items.Count > 0) {
    GUIArtesanosModificaciones dialogBox = new GUIArtesanosModificaciones(2);
    int indice = (this.ArtesanosListView.SelectedIndices.Count > 0) ?
                                        this.ArtesanosListView.SelectedIndices[0] : 0;
```

```
        dialogBox.Indice = indice;
        dialogBox.ListaArtesanos = this.listaArtesanos;
        dialogBox.Text = "Modificación de artesanos";
        //Si ya se presiono el boton OK del formulario anterior...
        if (dialogBox.ShowDialog(this) == DialogResult.OK){
            this.listaArtesanos = dialogBox.ListaArtesanos;
            //se muestran los datos
            this.LlenarArtesanosListView();
        }
        dialogBox.Dispose();
    } else {
        //Mostrar Cuadro de mensaje indicando que no existen datos
    }
```

Los métodos que forman parte de una modificación es el siguiente:

5.5.2.2.4.1 *Método EjecutarButton_Click*.

Lo que realiza este método es modificar los datos de la lista genérica de acuerdo al índice, y también realiza lo mismo para cada uno de los objetos de la base de datos que son extraídos para finalmente realizar los cambios de forma permanente en la base de datos. El código asociada a esta función es la siguiente:

```
using (TransactionScope scope = new TransactionScope())
{
    try
    {
        //Código asociado en la obtención de los datos de controles
        //Si se va a Insertar
        if (operacion == 1) {
            //Código asociado a una inserción
        } else if (operacion == 2) //Si se va a modificar
        {
            BDArteOnLineDataContext instancia = new BDArteOnLineDataContext();
            //se actualiza la lista listaArtesanos, que será devuelta al
            //formulario que lo instanció
            this.listaArtesanos[this.indice].Nombre = nombreArtesanoNuevo;
            this.listaArtesanos[this.indice].Apellidos = apellidosArtesanoNuevo;
            this.listaArtesanos[this.indice].Calle = calleArtesanoNuevo;
            this.listaArtesanos[this.indice].Ciudad = ciudadArtesanoNuevo;
            this.listaArtesanos[this.indice].Estado = estadoArtesanoNuevo;
            this.listaArtesanos[this.indice].Telefono = telefonoArtesanoNuevo;
            this.listaArtesanos[this.indice].Email = emailArtesanoNuevo;
            this.listaArtesanos[this.indice].Pseudonimo =
                                                    pseudonimoArtesanonuevo;
            this.listaArtesanos[this.indice].FechaModificacion =
                                                    this.FechaActual;

            //Modificar al objeto extraído dela BD
            Artesano actualizarBDArtesanos = (from q in instancia.Artesanos
                                              where q.IDArtesano ==
Int32.Parse(this.IDArtesanoTextBox.Text)
                                              select q).Single();
            actualizarBDArtesanos.Nombre = nombreArtesanoNuevo;
            actualizarBDArtesanos.Apellidos = apellidosArtesanoNuevo;
            actualizarBDArtesanos.Calle = calleArtesanoNuevo;
            actualizarBDArtesanos.Ciudad = ciudadArtesanoNuevo;
            actualizarBDArtesanos.Estado = estadoArtesanoNuevo;
            actualizarBDArtesanos.Telefono = telefonoArtesanoNuevo;
            actualizarBDArtesanos.Email = emailArtesanoNuevo;
            actualizarBDArtesanos.Pseudonimo = pseudonimoArtesanonuevo;
            actualizarBDArtesanos.FechaModificacion = this.FechaActual;
            if (CambioImagen) {
                imagenNueva = Conversiones.ImagenABytes(this.ImagenPictureBox.Image,
                                                        ImageFormat.Jpeg);
                this.listaArtesanos[this.indice].Imagen = imagenNueva;
                actualizarBDArtesanos.Imagen = imagenNueva;
            }
            //se actualiza hacia la BD
```

```
            instancia.SubmitChanges();
            scope.Complete();
            scope.Dispose();
            MessageBox.Show("Los artesanos fueron agregados existosamente", "Inserción",
                            MessageBoxButtons.OK, MessageBoxIcon.Information);
        }
    } catch (SqlException) {
        //Mostrar Cuadro de mensaje de error
    } catch (FormatException) {
        //Mostrar Cuadro de mensaje de datos incorrectos o campos vacíos
    }

}
```

5.5.2.2.5 Eliminación de artesanos.

En la interfaz *GUIArtesanos* el método encargado de este proceso es eliminarToolStripMenuItem_Click, **con el que** se eliminan uno o varios artesanos de la lista genérica, de la lista en el *ListView* y de la base de datos. El proceso de eliminado se realiza en esta misma interfaz por lo que no interviene la interfaz *GUIArtesanosModificaciones*. En este método también se comprueba si el artesano está asociado en las ordenes que haya realizado un cliente a través de la pagina web de *ArteOnLine*. La lógica implementada en este método es el siguiente:

```
ListView.SelectedIndexCollection coleccion = this.ArtesanosListView.SelectedIndices;
List<Artesano> listaArtesanosEliminar = new List<Artesano>();

//Mostrar Cuadro de mensaje indicando que se van a eliminar datos asignado a "resultado"

int cantidad = coleccion.Count;
if (cantidad > 0){
    if (resultado == DialogResult.Yes){
        try
        {
            BDArteOnLineDataContext bd = new BDArteOnLineDataContext();
            Artesano query = null;
            int numeroOrdenes = 0;
            int indiceColeccion = 0;
            //Se previene que el listView sea dibujado cada vez que se eliminan los elementos
            //que contiene, así primero se eliminan los objetos
            //y posteriormente se muestra el listView con los resultados finales
            this.ArtesanosListView.BeginUpdate();
            for (int i = 0; i < cantidad; i++) {
                //Para borrar solo los que no están asociados a una orden realizada
                //por un cliente
                numeroOrdenes = (from a in bd.Artesanos
                                 join b in bd.Artesanias on a.IDArtesano equals b.IDArtesano
                                 join c in bd.DetalleOrdens on b.IDArtesania
                                                            equals c.IDArtesania
                                 where a.IDArtesano ==
                                       listaArtesanos[coleccion[indiceColeccion]].IDArtesano
                                 select c.IDDetallePedido).Count();
                //Evaluar si los artesanos a eliminar tiene asociados ordenes
                if (numeroOrdenes > 0){
                    indiceColeccion++;
                    //se continua con el ciclo, para proseguir con los otros elementos
                    continue;
                } else {
                   //Si no se tienen asociados ordenes, entonces se borran los elementos de la
                    //base de datos. Primero se accede ala BD y se selecciona  al artesano de
                    //acuerdo al IDArtesano y se agrega a la lista listaArtesanosEliminar una
                    //por una
                    query = (from q in bd.Artesanos
                             where q.IDArtesano ==
                                      listaArtesanos[coleccion[indiceColeccion]].IDArtesano
                             select q).Single();
                    //Luego se eliminan los elementos tanto del listView (ArtesanosListView)
```

```
                    //como  de la lista listaArtesanos.
                    this.listaArtesanos.RemoveAt(coleccion[indiceColeccion]);
                    listaArtesanosEliminar.Add(query);
                    //Se eliminan los elementos del listview
                    this.ArtesanosListView.Items.RemoveAt(coleccion[indiceColeccion]);
                }
            }
            //se agregan todas las entidades en estado pending delete a través del
            //método DeleteAllOnSubmit
            bd.Artesanos.DeleteAllOnSubmit(listaArtesanosEliminar);
            //Se efectua la eliminación de los objetos en la BD
            bd.SubmitChanges();
            bd.Dispose();
            //se vuelve a dibujar el listView para mostrar los datos
            this.ArtesanosListView.EndUpdate();
            MessageBox.Show("Se han eliminado los artesanos de la base de datos",
                        "Eliminación",MessageBoxButtons.OK, MessageBoxIcon.Information);
            this.EstadoStrip.Text = "Se han eliminado los artesanos de la base de datos";
        } catch (SqlException ex){
            this.ArtesanosListView.EndUpdate();
            //Mostrar Cuadro de mensaje de error
        }
    }
}else{
    //Mostrar Cuadro de mensaje sin selección
}
```

5.5.2.2.6 Búsqueda de Artesanos.

El método que inicia este proceso en la interfaz *GUIArtesanos* es el método BuscarToolStripMenuItem_Click, que carga la interfaz *GUIBuscarArtesanos* para realizar la búsqueda de los artesanos desde la BD. El código definido para este método es el siguiente:

```
GUIBuscarArtesanos busquedaArtesanosForm = new GUIBuscarArtesanos();
busquedaArtesanosForm.ShowDialog(this);
if (busquedaArtesanosForm.DialogResult == DialogResult.OK) {
    this.listaArtesanos = busquedaArtesanosForm.ListaArtesanos;
    if (this.listaArtesanos.Count < 1) {
        //Mostrar Cuadro de mensaje sin resultados
    }
    this.LlenarArtesanosListView();
}
```

Esta interfaz *GUIBuscarArtesano* se muestra como un cuadro de dialogo en cuyo único campo vacío se puede agregar un parámetro para pode buscar sobre la base de datos al artesano de acuerdo a su nombre. El código definido en esta interfaz es la siguiente:

```
private List<ArtesanosLazy> listaArtesanos = new List<ArtesanosLazy>();
public List<ArtesanosLazy> ListaArtesanos {get { return this.listaArtesanos; }
 set { this.listaArtesanos = value; }
}

private void BuscarButton_Click(object sender, EventArgs e) {
    if (String.IsNullOrEmpty(this.BusquedaTextBox.Text)) {
        //Mostrar Cuadro de mensaje de campos vacíos
        return;
    }
    try {
        BDArteOnLineDataContext bd = new BDArteOnLineDataContext();
        //Forzar la ejecución inmediata con .ToLis();
        List<ArtesanosLazy> query = (from artesano in bd.Artesanos
                                     where
                                     artesano.Nombre.Contains(this.BusquedaTextBox.Text)
                                     ||
                                     artesano.Apellidos.Contains(this.BusquedaTextBox.Text)
                                     select new ArtesanosLazy {
                                         IDArtesano = artesano.IDArtesano,
```

```
                                        Nombre = artesano.Nombre,
                                        Apellidos = artesano.Apellidos,
                                        Calle = artesano.Calle,
                                        Ciudad = artesano.Ciudad,
                                        Estado = artesano.Estado,
                                        Telefono = artesano.Telefono,
                                        Email = artesano.Email,
                                        Pseudonimo = artesano.Pseudonimo,
                                        FechaRegistro = artesano.FechaRegistro,
                                        FechaModificacion =
                                                artesano.FechaModificacion,
                                        Imagen = artesano.Imagen
                                    }).ToList();
            listaArtesanos = query.GetRange(0, query.Count);
            query.Clear();
            bd.Dispose();
            this.DialogResult = DialogResult.OK;
        } catch (SqlException) {
            this.DialogResult = DialogResult.No;
            //Mostrar Cuadro de mensaje de error
        }
    }
```

5.5.2.3 Artesanías.

Para el caso de las artesanías, las interfaces *GUIArtesanias*, *GUIArtesaniasModificaciones*, *GUIBUscarArtesanias* tienen el mismo funcionamiento. Esto quiere decir que *GUIArtesanias* se encarga de cargar y mostrar los datos de las artesanías, del proceso de eliminación de la base de datos y además de mostrar las otras dos interfaces: *GUIArtesaniasModificaciones* y *GUIBUscarArtesanias*. Por su parte GUIArtesaniasModificaciones se encarga de las operaciones de inserción y modificación, y por último *GUIBuscarArtesanias* se encarga de la búsqueda de artesanías a través de su nombre.

5.5.2.3.1 Interfaz GUIArtesanias.

Los métodos que servirán para cualquier operación se enlistan a continuación:

5.5.2.3.1.1 *Método GUIArtesanias_Load*.

Este método cargar la información desde la base de datos de las artesanías, sin embargo además de cargar estos datos, también carga los datos de las clasificaciones de las artesanías y de los artesanos, ya que servirán para una inserción ó modificación y estos datos serán pasados a la interfaz *GUIArtesaniasModificaciones*.

```
//Cargar datos de los artesanos y clasificaciones (esto se hace por si se
//han agregado anteriormente nuevos artesanos y clasif)
BDArteOnLineDataContext cargarDatosBD = new BDArteOnLineDataContext();
//se llenan las listas
listaArtesanos = (from q in cargarDatosBD.Artesanos select q).ToList();
listaClasificaciones = (from q in cargarDatosBD.ClasificacionArtesanias
                        select
                            q).ToList();
//Inicializar el listView
this.InicializarArtesaniasListView();
this.todosToolStripMenuItem_Click(null, null);
cargarDatosBD.Dispose();
```

En el código mostrado anteriormente, se usa un método llamado *InicializarArtesaniasListView*, este inicializa el *ListView* en esta interfaz para que los datos se puedan observar en una lista, dado que anteriormente ya se mostró uno similar en el caso de los artesanos, el funcionamiento es el mismo para este método, solo que está centrado en los campos que están definidos en la tabla de las artesanías, por lo que el código será omitido.

5.5.2.3.1.2 *Método TodosToolStripMenuItem_Click*.

Este método tiene definida la conexión hacia la base de datos y la extracción de los datos para posteriormente agregarlos al *ListView* y a una lista genérica cuyo tipo es ArtesaniasJoin. Esta clase da soporte para las operaciones del tipo join que se realizan cuando se extraen las artesanías, ya que también se requiere de la extracción de las clasificaciones y los artesanos a los cuales pertenece. La clase *ArtesaniasJoin* está definida con los campos *IDArtesania (int), IDClasificacion (int), IDArtesano (int), DescripcionArtesania (string), NombreArtesano (string), NombreClasificacion (string), Cantidad (int), Precio (decima), Imagen (Binary), DateTime (FechaAlta), FechaModificacion (DateTime?)* y *Comentarios (string)*.

```
BDArteOnLineDataContext bd = new BDArteOnLineDataContext();
//se extraen todos los registros de la BD. Forzar la ejecución inmediata con .ToLis();
List<ArtesaniasJoin> query = (from artesania in bd.Artesanias
                              join clasificacion in bd.ClasificacionArtesanias on
                              artesania.IDClasificacion equals clasificacion.IDClasificacion
                              join artesano in bd.Artesanos on artesania.IDArtesano
                                                    equals artesano.IDArtesano
                              select new ArtesaniasJoin {
                                  IDArtesania = artesania.IDArtesania,
                                  IDArtesano = artesania.IDArtesano,
                                  IDClasificacion = clasificacion.IDClasificacion,
                                  DescripcionArtesania = artesania.DescripcionArtesania,
                                  NombreArtesano = artesano.Nombre + " " +
                                                                    artesano.Apellidos,
                                  NombreClasificacion = clasificacion.Clasificacion,
                                  Cantidad = artesania.Cantidad,
                                  Precio = artesania.Precio,
                                  Imagen = artesania.Imagen,
                                  FechaAlta = artesania.FechaAlta,
                                  FechaModificacion = artesania.FechaModificacion,
                                  Comentarios = artesania.Comentarios
                              }).ToList();
listaArtesanias = query.GetRange(0, query.Count);
query.Clear();
bd.Dispose();
//se llena el listview
this.LlenarArtesaniasListView();
```

5.5.2.3.2 Interfaz GUIArtesaniasModificaciones.

En esta interfaz se muestran los datos de los artesanos a detalle para poder agregar o modificar los registros de la base de datos. Los métodos que se usan de forma común en esta interfaz son los métodos *MostrarDatosListaArtesania* y *CargarImagenButton_Click*. El primero, de igual forma, tiene un funcionamiento y definición similares que el presentado en la interfaz *GUIArtesanosModificaciones*: el de cargar los datos y la imagen que se encuentran en los registros de la lista genérica y pasarlos hacia cada uno de los campos correspondientes de la interfaz *GUIArtesaniasModificaciones*, por este motivo, el código no es mostrado. El segundo método, su código también es omitidodao que el funcionamiento es el mismo que el presentado en la interfaz *GUIArtesanosModificaciones*: opera sobre el cambio de una imagen a partir de un archivo de imagen válido para un artesano.

5.5.2.3.3 Inserción de artesanías.

5.5.2.3.3.1 *Método AgregarToolStripMenuItem_Click*.

Este método en la interfaz *GUIArtesanias* , al igual que su homóloga presentada en la sección de artesanos, tiene asignada la función de cargar la interfaz para modificar los datos, pero esta vez la interfaz es *GUIArtesaniasModificaciones*. Dado que presenta la misma lógica de funcionamiento en el código que su homóloga *AgregarToolStripMenuItem_Click* en la interfaz *GUIArtesanosModificaciones*, éste será omitido.

5.5.2.3.3.2 *Método GUIArtesaniasModificaciones_Load*.

Este método en la interfaz GUIArtesaniasModificaciones carga los datos de las listas que contienen información tanto de los artesanos como de las clasificaciones, además de la información de las artesanías de la lista hacia los campos para que el usuario observe la información. El funcionamiento de este método se puede ver en el siguiente código:

```
this.fechaActual = DateTime.Now;
//Llenar los ComboBox con los datos de artesanos y clasificaciones
foreach (Artesano artesano in listaArtesanos) {
    this.ArtesanosComboBox.Items.Add(artesano.Nombre + " " + artesano.Apellidos);
}
foreach (ClasificacionArtesania clasificacion in listaClasificaciones) {
    this.ClasificacionesComboBox.Items.Add(clasificacion.Clasificacion);
}
//para saber a partir de que indice se va a insertar las nuevas artesanias en
//la lista listaArtesanias
this.elementosListaArtesanias = this.listaArtesanias.Count;
//Si se va a insertar
if (operacion == 1) {
    this.FechaAltaTextBox.Text = fechaActual.ToLongDateString();
} else if (operacion == 2)//Si se va a modificar {
    //Código asociado a una modificación
}
```

5.5.2.3.3.3 *Método EjecutarButton_Click*.

Este método tiene el mismo comportamiento e implementación de código que el método del mismo nombre presentado en la interfaz *GUIArtesanosModificaciones*. También, en esta parte se obtienen los datos de los controles y son evaluados para comprobar que tengan valores, posteriormente se insertan las nuevas artesanías tanto en la base de datos como en la lista que se muestran al usuario.

```
using (TransactionScope scope = new TransactionScope())
{
    try
    {
        int idArtesanoNuevo = this.listaArtesanos[idArtesanoAInsertar].IDArtesano;
        //se obtiene el id de la clasificacion de acuerdo al seleccionado en el comboBox
        int idClasificacionNueva =
this.listaClasificaciones[idClasificacionAInsertar].IDClasificacion;
        //lo demás se obtiene de los controles
        //string descripcionArtesaniaNueva, string nombreArtesanoNueva
        //string nombreClasificacionNueva, int cantidadNueva
        //Decimal precioNueva, Binary imagenNueva
        //string comentariosNueva

        if (descripcionArtesaniaNueva == null || cantidadNueva == 0 ||
            precioNueva == 0 || idClasificacionNueva < 1) {
            //Mostrar Cuadro de mensaje de campos vacíos
            return;
        }

        //Si se va a Insertar
        if (operacion == 1) {
            if (cambioImagen) { imagenNueva =
Conversiones.ImagenABytes(this.ImagenPictureBox.Image, ImageFormat.Jpeg); }
            else { imagenNueva =
Conversiones.ImagenABytes(Conversiones.RedimensionarImagen(Image.FromFile(rutaImagenes +
@"\Imagen no Disponible.jpg"), 100, 100, ImageFormat.Jpeg), ImageFormat.Jpeg); }
            //Instanciar para agregarlo a la lista
            ArtesaniasJoin nuevaArtesaniaJoin = new ArtesaniasJoin() {
                //a las propiedades se le asignan valores para actualizar la lista
                IDArtesano = idArtesanoNuevo,
                DescripcionArtesania = descripcionArtesaniaNueva,
                Cantidad = cantidadNueva,
                Precio = precioNueva,
                IDClasificacion = idClasificacionNueva,
                Imagen = imagenNueva,
```

```
                FechaAlta = this.fechaActual,
                NombreArtesano = nombreArtesanoNueva,
                NombreClasificacion = nombreClasificacionNueva,
                Comentarios = comentariosNueva
            };
            //Instanciar para agregarlo a la BD
            Artesania artesaniaNueva = new Artesania {
                IDArtesano = idArtesanoNuevo,
                DescripcionArtesania = descripcionArtesaniaNueva,
                Cantidad = cantidadNueva,
                Precio = precioNueva,
                IDClasificacion = idClasificacionNueva,
                Imagen = imagenNueva,
                FechaAlta = this.fechaActual,
                Comentarios = comentariosNueva
            };
            //se inserta la lista para agregarla a la BD
            this.listaArtesaniasInsertar.Add(artesaniaNueva);
            //Se agrega la artesania ala lista para actualizarla
            this.listaArtesanias.Add(nuevaArtesaniaJoin);

            this.cantidad++;
            this.IndiceToolStripTextBox.Text = cantidad.ToString();
            this.CantidadToolStripTexBox.Text = cantidad.ToString();
            // Se limpian los controles …

            this.FechaAltaTextBox.Text = this.fechaActual.ToLongDateString();
        } else if (operacion == 2) //Si se va a modificar
        {
            //Código asociado a una modificación
        }
    } catch (SqlException) {
        //Mostrar Cuadro de mensaje de error
    } catch (FormatException) {
        //Mostrar Cuadro de mensaje de datos incorrectos o campos vacíos
    }
}
```

5.5.2.3.3.4 *Método OKButton_Click*.

Este método presenta la misma estructura y lógica que la mostrada en la interfaz *GUIArtesaniasModificaciones*, por lo que el funcionamiento es el mismo. En este método tiene el siguiente código definido:

```
//si es insertar
using (TransactionScope scope = new TransactionScope())
{
try {
    if (operacion == 1) {
        int registros = this.listaArtesaniasInsertar.Count;
        if (registros > 0) {
          BDArteOnLineDataContext instanciaInsertar = new BDArteOnLineDataContext();
            for (int i = 0; i < registros; i++) {
                this.listaArtesanias[this.elementosListaArtesanias].IDArtesania =
                       instanciaInsertar.sp_CrearArtesania(
                                    listaArtesaniasInsertar[i].IDArtesano,
                                    listaArtesaniasInsertar[i].DescripcionArtesania,
                                    listaArtesaniasInsertar[i].Cantidad,
                                    listaArtesaniasInsertar[i].Precio,
                                    listaArtesaniasInsertar[i].IDClasificacion,
                                    listaArtesaniasInsertar[i].Imagen,
                                    listaArtesaniasInsertar[i].FechaAlta );
                this.elementosListaArtesanias++;
            }
            this.listaArtesaniasInsertar.Clear();
            scope.Complete();
            instanciaInsertar.Dispose();
            scope.Dispose();
            MessageBox.Show("Las artesanías fueron agregadas existosamente", "Inserción",
                         MessageBoxButtons.OK, MessageBoxIcon.Information);
```

```
            }
        }
        this.DialogResult = DialogResult.OK;
    } catch (SqlException) {
        this.DialogResult = DialogResult.No;
        //Mostrar Cuadro de mensaje de error
    }
    }
```

Se puede observar que se hace uso de un método llamado *sp_CrearArtesania*. Este método es el encargado de llamar a un procedimiento almacenado definido en SQL Server, el cual tiene el mismo nombre y su definición es la siguiente:

```
create procedure sp_CrearArtesania(
        @idArtesano int,
        @descripcionArtesania nvarchar(150),
        @cantidad int,
        @precio decimal(10,2),
        @idClasificacion int,
        @imagen varbinary(Max),
        @fechaAlta DateTime
)
As
Begin
DECLARE @IdArtesania int;
BEGIN TRANSACTION Artesania
INSERT INTO Artesania(IDArtesano, DescripcionArtesania, Cantidad, Precio,
                      IDClasificacion, Imagen,FechaAlta)
VALUES(@idArtesano, @descripcionArtesania, @cantidad, @precio,
                    @idClasificacion, @imagen,@fechaAlta);

        IF(SELECT @@Error)!=0
                Begin
                        RollBack Transaction Artesania;
                        Set @IdArtesania=0;
                End
        Else
                Begin
                        Commit Transaction Artesania;
                        Set @IdArtesania=@@Identity;
        End

return @IdArtesania;
End
```

Como se observa, este procedimiento recibe como parámetros todos los datos que son requeridos para la inserción de una artesanía, una vez que se obtienen los datos se inserta la artesanía en la base de datos, y a través de la sentencia *set @IdArtesania=@@Identity* se obtiene el valor del *Identity* que tiene asignado el campo *IDArtesania* de la tabla *Artesano* para finalmente ser devuelto por el procedimiento almacenado. Para el caso del método que es el resultado del mapeo del procedimiento almacenado a través del diseñador en Visual Studio 2008, no será presentado, ya que su definición es similar a la presentada en la sección 5.5.2.2.3 (Inserción de Artesanos), pero se presenta variaciones en cuanto a número y tipos de datos que se requieren para la inserción de una artesanía.

5.5.2.3.4 Modificación de artesanías.

El método que inicia este proceso en la interfaz *GUIArtesanias* es *ModificarToolStripMenuItem_Click*, el cual carga a la interfaz *GUIArtesaniasModifiaciones* con los datos que se tienen (lista de las artesanías, lista de las clasificaciones y una lista de los artesanos) con esto se van a mostrar los datos en campos donde se pueden modificar los valores de los campos que tenga en la base de datos, para posteriormente efectuar los cambios sobre la base de datos. El código asociado a este procedimiento es el siguiente:

```
if (this.ArtesaniasListView.Items.Count > 0) {
    //se realiza una instancia de GUIArtesaniasModificaciones, se pasan las listas
    //de artesanos y clasifiacciones y el tipo de operacion (2 = modificar)
    GUIArtesaniasModificaciones dialogBox = new
         GUIArtesaniasModificaciones(this.listaArtesanos, this.listaClasificaciones, 2);
    int indice = (this.ArtesaniasListView.SelectedIndices.Count > 0) ?
                          this.ArtesaniasListView.SelectedIndices[0] : 0;
    dialogBox.Indice = indice;
    dialogBox.ListaArtesanias = this.listaArtesanias;
    dialogBox.Text = "Modificación de artesanías";
    //Si ya se presiono el boton OK del formulario anterior...
    if (dialogBox.ShowDialog(this) == DialogResult.OK) {
        this.listaArtesanias = dialogBox.ListaArtesanias;
        //se muestran los datos
        this.LlenarArtesaniasListView();
    }
    dialogBox.Dispose();
} else {
    //Mostrar Cuadro de mensaje sin datos a modificar
}
```

5.5.2.3.4.1 Método EjecutarButton_Click.

Este método trabaja de la misma forma que la presentada en la sección 5.5.2.2.4 (Modificación de Artesanos), en el método con el mismo nombre, esto quiere decir que modifica los datos de la lista que se muestra al usuario y los que corresponden a la base de datos. El código asociado es el siguiente:

```
using (TransactionScope scope = new TransactionScope())
{
    try
    {
        //Código asociado en la obtención de los datos de controles
        //Si se va a Insertar
        if (operacion == 1) {
            //Código asociado a una inserción
        } else if (operacion == 2) //Si se va a modificar
        {
            BDArteOnLineDataContext instancia = new BDArteOnLineDataContext();
            //se actualiza la  lista listaArtesanias, que será devuelta al
            //formulario que lo instanció
            this.listaArtesanias[this.indice].IDArtesano = idArtesanoNuevo;
            this.listaArtesanias[this.indice].NombreArtesano = nombreArtesanoNueva;
            this.listaArtesanias[this.indice].DescripcionArtesania =
                                                descripcionArtesaniaNueva;
            this.listaArtesanias[this.indice].Cantidad = cantidadNueva;
            this.listaArtesanias[this.indice].Precio = precioNueva;
            this.listaArtesanias[this.indice].FechaModificacion = this.fechaActual;
            this.listaArtesanias[this.indice].NombreClasificacion = nombreClasificacionNueva;
            this.listaArtesanias[this.indice].Comentarios = comentariosNueva;

            //Modificar al objeto extraido dela BD
            Artesania actualizarBDArtesanias = (from q in instancia.Artesanias
                                                where q.IDArtesania ==
                                                Int32.Parse(this.IDArtesaniaTextBox.Text)
                                                select q).Single();
            actualizarBDArtesanias.IDArtesano = idArtesanoNuevo;
            actualizarBDArtesanias.DescripcionArtesania = descripcionArtesaniaNueva;
            actualizarBDArtesanias.Cantidad = cantidadNueva;
            actualizarBDArtesanias.Precio = precioNueva;
            actualizarBDArtesanias.IDClasificacion = idClasificacionNueva;
            actualizarBDArtesanias.FechaModificacion = this.fechaActual;
            actualizarBDArtesanias.Comentarios = comentariosNueva;
            if (cambioImagen) {
                imagenNueva = Conversiones.ImagenABytes(this.ImagenPictureBox.Image,
                                                        ImageFormat.Jpeg);
                listaArtesanias[this.indice].Imagen = imagenNueva;
                actualizarBDArtesanias.Imagen = imagenNueva;
            }
            //se actualiza hacia la BD
```

```
            instancia.SubmitChanges();
            scope.Complete();
            scope.Dipose();
            MessageBox.Show("La artesanía fue modificada existosamente", "Modificación",
                        MessageBoxButtons.OK, MessageBoxIcon.Information);
        }
    } catch (SqlException) {
        //Mostrar Cuadro de mensaje de error
    } catch (FormatException) {
        //Mostrar Cuadro de mensaje de datos incorrectos o campos vacíos
    }
}
```

5.5.2.3.5 Eliminación de artesanías.

En la interfaz *GUIArtesanias* el método encargado de este proceso es el método *EliminarToolStripMenuItem_Click*, a través de este método primero se comprueba si las artesanías están asociadas en las ordenes que haya realizado un cliente a través de la pagina web de ArteOnLine, si es así, no se borra, en caso contrario, serán eliminadas de la lista genérica, de la lista en el *ListView* y de la base de datos. El proceso de eliminación es el siguiente:

```
ListView.SelectedIndexCollection coleccion = this.ArtesaniasListView.SelectedIndices;
List<Artesania> listaArtesaniasEliminar = new List<Artesania>();
//Mostrar Cuadro de mensaje indicando que se van a eliminar datos asignado a "resultado"

int cantidad = coleccion.Count;
if (cantidad > 0) {
    if (resultado == DialogResult.Yes) {
        try {
            BDArteOnLineDataContext bd = new BDArteOnLineDataContext();
            Artesania query = null;
            //saber si el artesano se encuentra en ordenes realizadas
            int numeroOrdenes = 0;
            //indice para la coleccion de elementos seleccionados
            int indiceColeccion = 0;
            this.ArtesaniasListView.BeginUpdate();
            for (int i = 0; i < cantidad; i++) {
                //Para borrar solo las que no estan asociadas a una orden realizada
                //por un cliente
                numeroOrdenes = (from a in bd.Artesanias
                                 join b in bd.DetalleOrdens on a.IDArtesania
                                                            equals b.IDArtesania
                                 where
                        a.IDArtesania == listaArtesanias[coleccion[indiceColeccion]].IDArtesania
                                 select b.IDDetallePedido).Count();
                //Evaluar si las artesanias a eliminar tiene asociados ordenes
                if (numeroOrdenes > 0) {
                    //se incrementa en uno el indice de coleccion, puesto que no se ha
                    //eliminado ningun elemento
                    indiceColeccion++;
                    //se continua con el ciclo, para proseguir con los otros elementos
                    continue;
                } else {
                    //Primero se accede ala BD y se selecciona  la artesania de acuerdo al
                    //IDArtesania y se agrega a la lista listaArtesaniasEliminar una por una
                    query = (from q in bd.Artesanias
                             where q.IDArtesania ==
                            listaArtesanias[coleccion[indiceColeccion]].IDArtesania
                             select q).Single();
                    //Luego se eliminan los elementos tanto del listView (ArtesaniasListView)
                    //como  de la lista listaArtesanias.
                    this.listaArtesanias.RemoveAt(coleccion[indiceColeccion]);
                    listaArtesaniasEliminar.Add(query);
                    this.ArtesaniasListView.Items.RemoveAt(coleccion[indiceColeccion]);
                }
            }
            bd.Artesanias.DeleteAllOnSubmit(listaArtesaniasEliminar);
            //Se efectua la eliminación de los objetos en la BD
```

```
            bd.SubmitChanges();
            bd.Dispose();
            this.ArtesaniasListView.EndUpdate();
            MessageBox.Show("Se han eliminado las artesanías de la base de datos",
                        "Eliminación",MessageBoxButtons.OK, MessageBoxIcon.Information);
            this.EstadoStrip.Text = "Se han borrado las artesanías";

        } catch (Exception ex) {
            //Mostrar Cuadro de mensaje de error
        }
    } else {
        //Mostrar Cuadro de mensaje de sin selección de datos
    }
}
```

5.5.2.3.6 Búsqueda de Artesanías.

El método *buscarToolStripMenuItem_Click* en la interfaz *GUIArtesanias* inicia este proceso y carga la interfaz *GUIBuscarArtesanias* para realizar la búsqueda de las artesanías desde la base de datos. El patrón de implementación usado es el mismo que el presentado en la sección 5.5.2.2.6 (Búsqueda de artesanos), solamente que centrado a las artesanías. El código asociado es el siguiente:

```
GUIBuscarArtesanias busquedaArtesaniasForm = new GUIBuscarArtesanias();
busquedaArtesaniasForm.ShowDialog(this);
if (busquedaArtesaniasForm.DialogResult == DialogResult.OK) {
  this.listaArtesanias = busquedaArtesaniasForm.ListaArtesanias;
  if (this.listaArtesanias.Count < 1) {
     //Mostrar Cuadro de mensaje sin resultados
  }
  this.LlenarArtesaniasListView();
}
```

Por otro lado, el código que se encuentra en la interfaz *GUIBuscarArtesanias* y que realiza la búsqueda de las artesanías en la base de datos es el siguiente:

```
private List<ArtesaniasJoin> listaArtesanias = new List<ArtesaniasJoin>();
public List<ArtesaniasJoin> ListaArtesanias
{   get { return this.listaArtesanias; }
    set { this.listaArtesanias = value; } }

private void BuscarButton_Click(object sender, EventArgs e)
{
    if (String.IsNullOrEmpty(this.BusquedaTextBox.Text)) {
        //Mostrar Cuadro de mensaje de campo vacío
        return;
    }
    try
    {
        BDArteOnLineDataContext bd = new BDArteOnLineDataContext();
        List<ArtesaniasJoin> query = (from artesania in bd.Artesanias
                                      where
                    artesania.DescripcionArtesania.Contains(this.BusquedaTextBox.Text)
                                      join clasificacion in bd.ClasificacionArtesanias on
                                      artesania.IDClasificacion equals
                                                    clasificacion.IDClasificacion
                                      join artesano in bd.Artesanos on artesania.IDArtesano
                                                    equals artesano.IDArtesano
                                      select new ArtesaniasJoin
                                      {
                                          IDArtesania = artesania.IDArtesania,
                                          IDArtesano = artesania.IDArtesano,
                                          IDClasificacion = clasificacion.IDClasificacion,
                                          DescripcionArtesania =
                                                    artesania.DescripcionArtesania,
                                          NombreArtesano = artesano.Nombre + " " +
```

```
                                                        artesano.Apellidos,
                                        NombreClasificacion = clasificacion.Clasificacion,
                                        Cantidad = artesania.Cantidad,
                                        Precio = artesania.Precio,
                                        Imagen = artesania.Imagen,
                                        FechaAlta = artesania.FechaAlta,
                                        FechaModificacion = artesania.FechaModificacion,
                                        Comentarios = artesania.Comentarios
                                    }).ToList();
            this.listaArtesanias = query.GetRange(0, query.Count);
            query.Clear();
            bd.Dispose();
            this.DialogResult = DialogResult.OK;
        } catch (SqlException) {
            this.DialogResult = DialogResult.No;
            //Mostrar Cuadro de mensaje de error
        }
    }
```

5.5.2.4 Clientes.

Para poder observar los datos de los clientes que se encuentran en la base de datos, se hace uso de 3 interfaces, las cuáles son: *GUIClientes*, *GUIClientesDetalles* y *GUIBuscarClientes*. La primera interfaz, *GUIClientes*, es la encargada de cargar y mostrar los datos de los clientes que se hayan registrado en la página web de ArteOnLine, en la segunda interfaz, *GUIClientesDetalles* corresponde a la navegación sobre la lista que contiene a los clientes para observarlos a detalle uno por uno. Por último, la interfaz *GUIBuscarClientes* simplemente realiza una búsqueda sobre la tabla que contiene los datos de los clientes de acuerdo a su nombre y a un valor dado por el usuario para el mismo nombre.

En la interfaz *GUIClientes* se usa un método llamado *InicializarArtesaniasListView*, este inicializa el *ListView* en esta interfaz para que los datos se puedan observar en una lista, dado que anteriormente ya se mostró uno similar en el caso de los artesanos, el funcionamiento es el mismo para este método, solo que está centrado en los campos que están definidos en la tabla de los clientes, por lo que el código será omitido. Los métodos que servirán para cualquier operación se enlistan a continuación:

5.5.2.4.1 Interfaz GUIClientes.

5.5.2.4.1.1 *Método todosToolStripMenuItem_Click*.

Este método se encarga de extraer solamente los datos importantes de los clientes que se hayan registrado en la página web de ArteOnLine y posteriormente mostrarlos sobre una lista. El código de este método es el siguiente:

```
BDArteOnLineDataContext bd = new BDArteOnLineDataContext();
//Forzar la ejecución inmediata con .ToLis();
List<ClienteJoin> query = (from cliente in bd.Clientes
                           join tipo in bd.TipoClientes
                           on cliente.IDTipoCliente equals tipo.IDTipoCliente
                           select new ClienteJoin
                           {
                               IDTipoCliente = tipo.IDTipoCliente,
                               IDCliente = cliente.IDCliente,
                               NombreTipoCliente = tipo.Tipo,
                               Nombre = cliente.Nombre,
                               Apellidos = cliente.Apellidos,
                               NombreUsuario = cliente.NombreUsuario,
                               Calle = cliente.Calle,
                               Ciudad = cliente.Ciudad,
                               Estado = cliente.Estado,
                               Telefono = cliente.Telefono,
                               Email = cliente.Email,
                               FechaRegistro = cliente.FechaRegistro,
                               FechaModificacion = cliente.FechaModificacion,
```

```
                                }).ToList();
listaClientes = query.GetRange(0, query.Count);
query.Clear();
bd.Dispose();
this.LlenarArtesanosListView();
```

Como se ha ido manejando en las interfaces que muestran la información sobre listas, tanto el método que inicializa el *ListView* (llamado *InicializarClientesListView*) para que los datos se puedan observar en una lista, como el método *LlenarArtesanosListView*, que carga la información de las listas sobre el *ListView*, son omitidos ya que se presntaron en las secciones anteriores unos identicos. También se usa una clase llamada *ClientesJoin*, ya que se extrae la información del tipo de cliente a través de una operación *join*, esta clase está definida con los siguientes campos: *IDTipoCliente (int), NombreTipoCliente (string), IDCliente (int), Nombre (string), Apellidos (string), NombreUsuario (string), Calle (string), Ciudad (string), Estado (string), Telefono (string), Email (string),FechaRegistro (DateTime)* y *FechaModificacion (DateTime?)*. De esta forma, solamente el nombre de usuario se puede observar, mas no así el password.

5.5.2.4.2 Interfaz GUIClientesDetalles.

5.5.2.4.2.1 *Método MostrarDatosListaClientes*.

Este método carga los datos que se encuentran en los registros de la lista genérica de los clientes (provenientes de la base de datos) y pasarlos hacia cada uno de los campos correspondientes a esta interfaz, por razones de espacio el código no es presentado ya que de igual manera tiene la misma lógica implementada como en los casos anteriores para los métodos de similar nombre en las interfaces como *GUIArtesanosModificaciones* en el método *MostrarDatosListaArtesano*.

5.5.2.4.3 Navegación sobre clientes.

El método *DetalleToolStripMenuItem_Click* en la interfaz *GUIClientes* muestra la interfaz *GUIClientesDetalles* y le porporciona una lista con los datos de los clientes extraídos de la base de datos. El código asociado a esta función es la siguiente:

```
if (this.ClientesListView.Items.Count > 0) {
    GUIClientesDetalles dialogBox = new GUIClientesDetalles();
    int indice = (this.ClientesListView.SelectedIndices.Count > 0) ?
                    this.ClientesListView.SelectedIndices[0] : 0;
    dialogBox.Indice = indice;
    dialogBox.listaClientes = this.listaClientes;
    dialogBox.ShowDialog(this);
} else {
    //Mostrar Cuadro de mensaje sin resultados
}
```

Por otro lado, en la interfaz *GUIClientesDetalles* se encuentra el método *GUIClientesDetalles_Load*, que se encarga de llamar a otro método para poder llenar los campos con los datos del primer elemento de la lista genérica de los clientes, para posteriormente navegar sobre cada uno de estos elementos. El código asociado es el siguiente:

```
    this.cantidad = this.listaClientes.Count;
    MostrarDatosListaClientes();
    this.menuStrip1.Enabled = true;
```

5.5.2.4.4 Navegación sobre órdenes de los clientes.

5.5.2.4.4.1 *Método OrdenesToolStripMenuItem_Click*.

Dado que es obvio que al encontrarse en la interfaz *GUIClientes* que muestra los datos de los clientes, también debe mostrarse los datos de las órdenes que hayan realizado. En este caso

existe otra interfaz que muestra las órdenes que existen en la base de datos sobre una lista, la cuál es denominada *GUIOrdenes*, esta interfaz será mostrada posteriormente, por el momento se mostrará el método que carga esta interfaz, método que lleva por nombre *OrdenesToolStripMenuItem_Click* y carga a la interfaz *GUIOrdenes* para cargar los datos de las ordenes desde la base de datos, sin embargo, solo se cargan las ordenes de cierto cliente (no todas las ordenes). El código es el siguiente:

```
if (this.ClientesListView.SelectedIndices.Count > 0) {
    int indice = this.ClientesListView.SelectedIndices[0];
    int idCliente = this.listaClientes[indice].IDCliente;
    string cliente = this.listaClientes[indice].Nombre + " " +
                     this.listaClientes[indice].Apellidos;

    GUIOrdenes ordenesForm = new GUIOrdenes();
    ordenesForm.WindowState = FormWindowState.Normal;
    ordenesForm.StartPosition = FormStartPosition.CenterScreen;
    ordenesForm.Text += " de: " + cliente;
    ordenesForm.SeleccionOrdenesTotales = false;
    ordenesForm.Cliente = cliente;
    ordenesForm.IdClienteOrden = idCliente;
    ordenesForm.MdiParent = this.ParentForm;

    ordenesForm.Show();
} else {
    //Mostrar Cuadro de mensaje sin selección de datos
}
```

5.5.2.4.4 Búsqueda de clientes.

El método en la interfaz *GUIClientes* que realiza esta función, es el método *buscarToolStripMenuItem_Click* carga la interfaz *GUIBuscarClientes* para realizar la búsqueda de los clientes desde la BD. El código definido sigue el mismo patrón que el mostrado en las interfaces *GUIBUscarArtesanos* y *GUIBuscarArtesanias* son la diferencia de que está centrado sobre la tabla de las órdenes, por lo que dicho código no será mostrado.

Posteriormente la interfaz *GUIBuscarClientes*, se muestra como un cuadro de dialogo en cuyo único campo vacío se puede agregar un parámetro para poder buscar sobre la base de datos al cliente de acuerdo a su nombre. El código que está asociado a esta interfaz y que solo realiza la operación de búsqueda no será mostrado por cuestiones de espacio pero sigue el mismo comportamiento que en las demás interfaces que realizan la búsqueda sobre las tablas de artesanía y artesano.

5.5.2.5 Órdenes.

Con respecto a las órdenes de los clientes en la página de ArteOnLine, se tienen 3 interfaces, la primera de ellas llamada *GUIOrdenes* (mencionada anteriormente), sirve para observar las órdenes sobre una lista y a partir de esta interfaz realizar la modificación o poder observar a detalle cada una de las órdenes (cliente, artesanías y sus precios). La segunda, *GUIDetalleOrden*, sirve para observar a detalle cada una de las órdenes que se muestran en la lista de la interfaz *GUIOrdenes* y la tercera, *GUIOrdenesModificaciones*, se encarga de modificar el status de cada orden.

5.5.2.5.1 *Interfaz GUIOrdenes*.

El código que se encarga directamente de la extracción de los datos desde la base de datos es el siguiente:

```
BDArteOnLineDataContext bd = new BDArteOnLineDataContext();
if (selectAll) {
    this.listaOrdenes = (from cliente in bd.Clientes
                         join orden in bd.Ordens on cliente.IDCliente
                                   equals orden.IDCliente
                         join status in bd.StatusOrdens on orden.IDStatusOrden
                                   equals status.IDStatusOrden
                         select new OrdenesJoin
                         {
                             IdPedido = orden.IDPedido,
                             FechaInicio = orden.FechaOrden,
                             FechaEntrega = orden.FechaEntrega,
                             Cliente = cliente.Nombre + " " + cliente.Apellidos,
                             Status = status.Status
                         }).ToList();

} else {
    this.listaOrdenes = (from orden in bd.Ordens
                         where orden.IDCliente == idCliente
                         join cliente in bd.Clientes on orden.IDCliente
                                    equals cliente.IDCliente
                         join status in bd.StatusOrdens on orden.IDStatusOrden
                                   equals status.IDStatusOrden
                         select new OrdenesJoin
                         {
                             IdPedido = orden.IDPedido,
                             FechaInicio = orden.FechaOrden,
                             FechaEntrega = orden.FechaEntrega,
                             Cliente = cliente.Nombre + " " + cliente.Apellidos,
                             Status = status.Status
                         }).ToList();
}
bd.Dispose();
```

La forma en cómo carga los datos de las ordenes difiere cuando se llama este método desde la misma interfaz *GUIOrdenes* que cuando se llama desde la interfaz *GUIClientes*, ya que en la primera interfaz se cargan todas las ordenes, mientras que en la segunda solo se cargan las ordenes de cierto cliente. Los resultados para cualquier caso son pasados a una lista genérica de tipo *OrdenesJoin*, esta clase esta creada para dar soporte al tipo de datos que se extraen de la base de datos, puesto que solo se extraen los datos más importantes de una orden, esta clase debe tener las mismas propiedades que los campos obtenidos de la consulta en LINQ. Esta clase tiene definido los siguientes campos: *IdPedido (int)*, *FechaInicio (DateTime)*, *FechaEntrega (DateTime)*, *Cliente (string)* y *Status (string)*.

De esta forma, si se carga esta interfaz desde la interfaz usada para los clientes (*GUIClientes*), el código que se usa para poder cargar los datos de un cliente es el mostrado en la sección ***Navegación sobre órdenes de los clientes***. Mientras que para cargar todas las órdenes se utiliza el siguiente código en la carga del formulario *GUIOrdenes*:

```
this.SeleccionarOrdenesClientes(true, 0);
```

5.5.2.5.2 Ver detalles de la orden realizada.

5.5.2.5.2.1 *Método DetallesToolStripMenuItem_Click*.

Este método en la interfaz *GUIOrdenes* se encarga de mostrar la interfaz *GUIDetalleOrden* para poder observar cuáles son las artesanías asociadas a una orden en específico que haya sido realizada por un cliente en la página web de ArteOnLine, así como las cantidades y precios (subtotales y total) que deberá pagar por dicha orden. El código contenido en este método es el siguiente:

```
if (this.listaOrdenes.Count > 0){
    if (this.OrdenesListView.SelectedIndices.Count > 0) {
        GUIDetalleOrden detalleOrdenesDialogBox = new GUIDetalleOrden();
```

```
        //SelectedIndices devuelve una colección ListView.SelectedIndexCollection
        //conteniendo los indices de todos los items seleccionados en el ListView.
        //Solo nos interesa el primer elemento (indice 0), ya que solo
        // se puede seleccionar un solo elemento en el ListView y no varios
        int indice = (this.OrdenesListView.SelectedIndices.Count > 0) ?
                                this.OrdenesListView.SelectedIndices[0] : 0;
        this.IdClienteOrden = Int32.Parse(this.OrdenesListView.Items[indice].Text);
        detalleOrdenesDialogBox.idPedido = this.IdClienteOrden;
        detalleOrdenesDialogBox.cliente =
                            this.OrdenesListView.Items[indice].SubItems[3].Text;
        detalleOrdenesDialogBox.ShowDialog(this);
    }
    else {
        //Mostrar Cuadro de mensaje sin selección de datos
    }
} else {
    //Mostrar Cuadro de mensaje sin resultados
}
```

5.5.2.5.2.2 *Interfaz GUIDetalleOrden*.

El método encargado específicamente de obtener la información detallada de una orden a través de operaciones *join* de la base de datos es *GUIDetalleOrden_Load*. La clase *DetalleOrdenJoin* utilizada para contener los datos que son extraídos tiene los siguientes campos: *Cliente (string), Atresania (string), PrecioUnitario (decimal), Subtotal (decimal), Cantidad (int)* y *FechaModificacion (DateTime?)*. El código definido para esta función es el siguiente:

```
BDArteOnLineDataContext bd = new BDArteOnLineDataContext();
this.InicializarDetalleOrdenListView();
this.listaDetalleOrden = (from detalleOrden in bd.DetalleOrdens
                          where detalleOrden.IDPedido == this.idPedido
                          join orden in bd.Ordens on detalleOrden.IDPedido
                                                  equals orden.IDPedido
                          join artesania in bd.Artesanias on detalleOrden.IDArtesania
                                                  equals artesania.IDArtesania
                          select new DetalleOrdenJoin
                          {
                              Artesania = artesania.DescripcionArtesania,
                              Cantidad = detalleOrden.Cantidad,
                              PrecioUnitario = detalleOrden.PrecioUnitario,
                              SubTotal = detalleOrden.Cantidad * detalleOrden.PrecioUnitario,
                              FechaModificacion = detalleOrden.FechaModificacion
                          }).ToList();
bd.Dispose();
this.LlenarDetalleOrdenListView();
```

Los métodos *InicializarDetalleOrdenListView* y *LlenarDetalleOrdenListView* los cuáles no son mostrados, son usados para definir y llenar el *ListView* respectivamente con los datos que son extraídos.

5.5.2.5.3 Modificar Status de la Orden.

El método encargado de realizar iniciar este proceso en la interfaz *GUOrdenes* es *ModificarToolStripMenuItem_Click*. En este método se muestra la interfaz *GUIOrdenModificaciones* en la cual se modificará solo el campo *StatusOrden* de una orden para que el estado de una orden pase por diferentes etapas, que se mencionan a continuación:

1. Iniciada, si la orden ha sido creada.
2. Entregada, si la orden ya ha se encuentra con el cliente.
3. Pendiente, si durante el proceso de atención de la orden (suministro de una artesanía), ocurre algún evento inesperado.
4. Procesando, si la orden ya está siendo atendida en cuanto al suministro de la(s) artesanía(s) que han sido solicitadas.

5. Enviada, si la orden ya ha sido liberada para que llegue al cliente.

Finalmente la interfaz es mostrada siguiendo la misma lógica cuando se muestra un cuadro de mensaje para modificación como lo es por ejemplo en los casos de las artesanías o artesanos. El código asociado al método *ModificarToolStripMenuItem_Click* es el siguiente:

```
if (this.OrdenesListView.Items.Count > 0) {
    if (this.OrdenesListView.SelectedIndices.Count > 0) {
        // SelectedIndices devuelve una colección ListView.SelectedIndexCollection
        //conteniendo los indices de todos los items seleccionados en el ListView.
        int indice = this.OrdenesListView.SelectedIndices[0];
        GUIOrdenModificaciones ordenModificacionesForm = new GUIOrdenModificaciones();
        this.IdClienteOrden = Int32.Parse(this.OrdenesListView.Items[indice].Text);
        ordenModificacionesForm.idOrden = this.IdClienteOrden;
        ordenModificacionesForm.status =
                                    this.OrdenesListView.Items[indice].SubItems[4].Text;
        if (ordenModificacionesForm.ShowDialog(this) == DialogResult.OK) {
            this.OrdenesListView.Items[indice].SubItems[4].Text =
                                                ordenModificacionesForm.status;
        } else {
            //Mostrar Cuadro de mensaje sin selección de datos
        }
    } else {
        //Mostrar Cuadro de mensaje sin ordenes a modificar
    }
}
```

Por otro lado, en la interfaz *GUIOrdenModificaciones* solo se muestra un solo control, un *ComboBox*, en el cuál se muestran todas las opciones existentes y que son válidas para modificar el estado de una orden, desde iniciada hasta entregada. Para evitar cargar los datos de la tabla *StatusOrden*, el cuál contiene todas las opciones mostradas anteriormente. Dado que estas opciones permanecen constantes, se opto por crear una fuente de datos para el *ComboBox* en tiempo de ejecución y así evitar cargar los datos desde la base de datos. El código en cargado de esto es el siguiente:

```
List<StatusOrden> listaStatus = new List<StatusOrden> {
              new StatusOrden{IDStatusOrden=1,Status="Iniciada"},
                 new StatusOrden{IDStatusOrden=2,Status="Entregada"},
                 new StatusOrden{IDStatusOrden=3,Status="Pendiente"},
                 new StatusOrden{IDStatusOrden=4,Status="Procesando"},
                 new StatusOrden{IDStatusOrden=5,Status="Enviada"}
                 };

this.StatusComboBox.DataSource = listaStatus;
this.StatusComboBox.DisplayMember = "Status";
this.StatusComboBox.ValueMember = "IDStatusOrden";
this.IdOrdenTextBox.Text = this.idOrden.ToString();
this.StatusComboBox.Text = this.status;
```

Como se puede observar, se crea una lista genérica con tipos *StatusOrden*, esta clase fue creada al mapear la tabla *StatusOrden* en Visual Studio, esta lista contendrá todos los posibles valores válidos para modificar el estado de una orden. Posteriormente esta fuente de datos es enlazada al *ComboBox* a través de la propiedad *DataSource*. Finalmente cuando se carga la interfaz, a un campo se le es asignada el id de la orden y al *ComboBox* también se le asigna el valor del estado que fueron asignados en la interfaz *GUIOrdenes*.

El siguiente método se encarga de realizar los cambios en el status de una orden sobre la base de datos, para lo cual el código es el siguiente:

```
using (TransactionScope scope = new TransactionScope()){
    try
    {
        BDArteOnLineDataContext actualizarOrden = new BDArteOnLineDataContext();
        Orden orden = (from o in actualizarOrden.Ordens
```

```
                    where o.IDPedido == this.idOrden select o).Single();
        //Seleccionando el detalle del pedido para modificar la fecha de modificación
        var detalleOrden = from d in actualizarOrden.DetalleOrdens
                           where d.IDPedido == orden.IDPedido select d;
        //Actualizando las fechas de modificacion para c/ude los productos
        //que hayan en el detalle del pedido
        foreach (var item in detalleOrden) {
            item.FechaModificacion = DateTime.Now;
        }
        orden.IDStatusOrden = (int)this.StatusComboBox.SelectedValue;
        //se actualiza la base de datos
        actualizarOrden.SubmitChanges();
        scope.Complete();
        //se libera la memoria asociada al objeto
        actualizarOrden.Dispose();
        this.status = this.StatusComboBox.Text;
        scope.Dispose();
        MessageBox.Show("Se ha modificado el estado de la orden exitosamente.",
                    "Modificación", MessageBoxButtons.OK, MessageBoxIcon.Information);
        this.DialogResult = DialogResult.OK;
    } catch (SqlException) {
        //Cuadro de mensaje de error
    }
}
```

5.5.3 DISEÑO DE LA APLICACIÓN WEB ASP.NET ARTEONLINE.

Esta aplicación WEB está diseñada para responder a las necesidades de la atención de los clientes a través de internet y exponer las artesanías de los artesanos que se registran en la base de datos con el fin de comercializarlos. En esta aplicación web se puede registrar un nuevo cliente, iniciar sesión para poder acceder a las características de la compra de las artesanías y la visión de las órdenes que hay realizado dicho cliente, entre otras funciones que se verán más adelante.

5.5.3.1 Mapeo de la base de datos ArteOnLine para la aplicación Web ASP.NET.

Antes de iniciar el desarrollo de la aplicación Web, previamente se debió haber mapeado la base de datos ArteOnLine en Visual Studio 2008 siguiendo los pasos mencionados anteriormente en la sección 5.5.1, pero esta vez para una aplicación web de *asp.net* y solo extrayendo los datos importantes de la base de datos que serán usados.

5.5.3.2 Modo de autenticación por formularios.

La autenticación de formularios habilita la validación de usuario y contraseña para aplicaciones web que no requieren la autenticación de Windows. En el siguiente fragmento de código ubicado en el archivo *web.config* de la aplicación se asigna la forma de autenticación que se usará a través en el contexto de la aplicación, la cuál es a través de los formularios y en donde se especifica cuál es el formulario de login (*login.aspx*) y en caso de realizar un *login* la página a la cuál se re direccionará una vez que se a logrado autenticar (*Default.aspx*). Con esto cualquier solicitud para una página ASP.NET que forma parte de la aplicación requiere que la autenticación de formularios se proporcione un nombre de usuario válido.

```
<authentication mode="Forms">
  <forms name="ArteOnline" loginUrl="login.aspx"
         defaultUrl="Default.aspx" timeout="15" protection="All" path="/"/>
</authentication>
<authorization>
  <allow users="*"/>
</authorization>
```

5.5.3.3 Restricción sobre directorios.

En el siguiente fragmento de código ubicado en el archivo *web.config* de la aplicación se establece el directorio en la raíz de la aplicación a la cual se restringirá el acceso y en la que se deberá estar autenticado para poder acceder a la ruta definida, la cuál es "finalizar" y dentro de este directorio se encuentran los formularios *carrito.aspx* y *seguimientoOrdenes.aspx*.

```
<location path="finalizar">
  <system.web>
    <authorization>
      <deny users="?"/>
    </authorization>
  </system.web>
</location>
```

5.5.3.4 Aspecto General.

Para poder simplificar el código *html* dentro de los formularios web de *asp.net* de la aplicación web *ArteOnLine*, se optará por mostrar primero las definiciones globales que tendrá cada una de los formularios encargados de trabajar en interactuar con el cliente y la base de datos a través del navegador de internet, es decir, las páginas default.aspx, registro.aspx, login.aspx, carrito.aspx y seguimientoOrdenes.aspx. El código html es el siguiente:

```
<head id="Head1" runat="server">
    <title>Arte Online</title>
    <link href="estilos/GridView.css" rel="stylesheet" type="text/css" />
    <link href="estilos/Body.css" rel="stylesheet" type="text/css" />
    <link href="estilos/TituloDIv.css" rel="stylesheet" type="text/css" />
    <link href="estilos/NuevoMenu.css" rel="stylesheet" type="text/css" />
    <link href="estilos/NuevoMenu2.css" rel="stylesheet" type="text/css" />
</head>
<body class="Cuerpo">
    <form id="form1" runat="server" style="font-family: Calibri; font-weight: bold; width:
970px;
    margin: 0px auto auto auto">
    <!--Definición del Div para el logotipo !-->
    <div class="TituloDiv">
        <img src="imagenes/LogoTipoTexturaLienzo.png" alt="Logo" style="width: 970px; height:
120px;" />
    </div>
    <!--Definicion del div que a su vez contiene 3 div más:
        El primero es para le sección donde se coloca la opción de busqueda de las
        Artesanías
        El segundo corresponde al menú que lleva hacia las diferentes páginas
        El tercero está dedicado al área espedífica del formulario aspx y es la que contiene
        los diferentes controles de asp.net !-->
    <div style="width: 970px; margin: 0px auto;">
        <!--div para la búsqueda !-->
        <div class="menu">
        <!--Definicion de la lista para la busqueda de artesanias !-->
            <ul>
        <!--Agregado de los listItem dentro de la lista desordenada ul !-->
                <li style="float: left;">
                    <asp:Label ID="BienvenidoLabel" runat="server" ForeColor="#000000" Font-
Names="Calibri"
                        Width="200px"></asp:Label>
                </li>
                <li style="float: right;">
                    <asp:Label ID="BuscarLabel" runat="server" Text="Buscar:"
Height="23px"></asp:Label>
                    <asp:TextBox ID="BuscarTextBox" runat="server"></asp:TextBox>
                    <asp:Button ID="BuscarButton" runat="server" Text="Buscar" Font-
Names="Calibri" OnClick="BuscarButton_Click" />
                    <asp:LinkButton ID="LogOutLinkButton" runat="server"
OnClick="LogOutLinkButton_Click"
                        Text="LogOut" Font-Underline="true" ForeColor="Black" Font-
Bold="true" Visible="true"></asp:LinkButton>
                </li>
```

```
            </ul>
        </div>
        <br />
        <!--div para el menú  !-->
        <div class="menu2">
            <ul>
                <li>
                    <!-- Se agrega un boton de ASP.NET para poder ejecutar una accion !-->
                    <asp:LinkButton ID="LinkButton1" runat="server"
OnClick="InicioLinkButton_Click"
                        Text="Artesanías"></asp:LinkButton>
                </li>
                <!-- Se agregan link's a otras paginas !-->
                <li><a href="finalizar/carrito.aspx">Carrito</a></li>
                <li><a href="login.aspx">Login</a></li>
                <li><a href="registro.aspx">Registro</a></li>
            </ul>
        </div>
        <!--div para los controles asp.net o elementos html !-->
        <div style="float: left; width: 830px">

         <!--Definición de la estructura y de los controles específicos para
             c/u de las páginas!-->

        </div>
    </div>
    <br style="clear: both" />
    <br />
    <div style="margin: 0px auto auto auto; width: 970px; text-align: center">
        2009 UNAM FES ARAGÓN
    </div>
    </form>
</body>
</html>
```

Como se observa en el código anterior en primera instancia se declaran los recursos con los que contará la página *html* dentro de la etiqueta *Head*, en este caso son archivos *css* que servirán para definir el aspecto de la página y de los elementos contenidos en ella, como por ejemplo la posición, color de fondo, aspecto de los elementos d*iv* que contienen a los demás elementos *html,* aspecto y definición de los elementos *ul* (listas desordenadas) y sus *listitem* (*li*). Posteriormente se declaran los elementos *div* que servirán para poder dividir la estructura de las páginas *html* y dentro de estos elementos (d*iv*) contener a los elementos *html* como tablas, botones, campos para la recepción de datos, *GridView* para poder mostrar las artesanías de la base de datos, de los elementos del carrito de compra, de las órdenes realizadas, etc. A partir de este momento, en las siguientes secciones solo se mostrará el código html específico de cada formulario *.aspx* para poder ahorrar espacio.

5.5.3.5 Registro de clientes (*registro.aspx*).

En esta página, como su nombre lo indica se realiza el proceso de inserción del cliente sobre la base de datos, para que posteriormente realizar un login e ingresar a la sección del carrito de compra y poder observar las órdenes que haya realizado, además de poder agregar elementos al carrito de compra. De no ser así cada vez que requiera observar las órdenes realizadas o desee agregar un ítem al carrito no podrá realizarlo, a menos que realice el login con su nombre de usuario y contraseña correspondientes. El código específico para la definición del formulario *registro.aspx* es el siguiente:

```
<!-- tabla exterior!-->
<table style="border-style: solid; border-color: #000000; width: 400px; height: auto;
    background-color: #B9AF6F" align="center">
    <tr>
        <td>
            <!-- tabla interior !-->
```

```
            <table style="width: 415px; background-color: #B9AF6F;" align="center">
                <tr>
                    <td>
                        <br />
                    </td>
                </tr>
                <tr style="width: 800px">
                    <!-- asp:TextBox Nombre de usuario!-->
                    <td> Nombre de usuario: </td>
                    <td> <asp:TextBox ID="UserNameTextBox" runat="server"
                                      Width="260px"></asp:TextBox>
                        <br />
                        <asp:RequiredFieldValidator ID="UserNameRequiredFieldValidator"
                         runat="server" ErrorMessage="El nombre de usuario no puede estar vacío"
                         ControlToValidate="UserNameTextBox"
                         Display="Dynamic"></asp:RequiredFieldValidator>
                    </td>
                </tr>
                <tr>
                    <!-- asp:TextBox Contraseña!-->
                    <td> Contraseña: </td>
                    <td> <asp:TextBox ID="PasswordTextBox" runat="server" TextMode="Password"
                                      Width="260px"></asp:TextBox>
                        <br />
                        <asp:RequiredFieldValidator ID="PasswordRequiredFieldValidator"
                                 runat="server" ErrorMessage="La contraseña no puede estar vacía"
                                 ControlToValidate="PasswordTextBox"
                                 Display="Dynamic"></asp:RequiredFieldValidator>
                    </td>
                </tr>
                <tr>
                    <!-- asp:TextBox Confirmar Contraseña  !-->

                    <!-- asp:TextBox Nombre !-->
                    <td> Nombre: </td>
                    <td> <asp:TextBox ID="NombreTextBox" runat="server"
                                      Width="260px"></asp:TextBox>
                        <br />
                        <asp:RadioButton ID="PersonaRadioButton" runat="server" Text="Persona"
                                         Checked="True" GroupName="TipoCliente"
                                         OnCheckedChanged="PersonaRadioButton_CheckedChanged" />
                        <asp:RadioButton ID="AsociacionRadioButton" runat="server"
                                          Text="Asociación" GroupName="TipoCliente"
                                          OnCheckedChanged="AsociacionRadioButton_CheckedChanged" />
                        <br />
                        <asp:RequiredFieldValidator ID="NombreRequiredFieldValidator"
                                    runat="server" ErrorMessage="El nombre no puede estar vacío"
                                    ControlToValidate="NombreTextBox"
                                    Display="Dynamic"></asp:RequiredFieldValidator>
                    </td>
                </tr>
                <tr>
                    <!-- asp:TextBox Apellidos !-->
                    <!-- asp:TextBox Calle !-->
                    <!-- asp:TextBox Ciudad !-->
                    <!-- asp:TextBox Estado !-->
                    <!-- asp:TextBox Teléfono !-->
                    <!-- asp:TextBox Email !-->
                    <!-- boton asp.net aceptar !-->
                </tr>
            </table>
        </td>
    </tr>
</table>
```

Se observa que dentro de la etiqueta *body* se define una tabla que contendrá elementos propios de asp.net como son los controles de validación *asp:RequiredFieldValidator* asociados a los controles de servidor de entrada como lo son los *TextBox*, los cuáles verifican que existan datos escritos en el control *TextBox*, ya que no permiten avanzar hasta que tener cierto valor escrito, es

decir que convierten el campo en obligatorio. Estos elementos corresponden a los datos que se requieren para que un cliente pueda registrarse como lo es el nombre de usuario, password, nombre, apellidos, calle, etc. Por su parte, el código asociado a la función de inserción (*code-behind)* es la siguiente:

```
protected void RegistrarButton_Click(object sender, EventArgs e) {
    if (this.PasswordTextBox.Text != this.RePasswordTextBox.Text){
        this.ErrorLabel.Text = "Las contraseñas no coinciden, verifíquelas.";
    }
    else{
        try {
            ArteOnlineDataContext bd = new ArteOnlineDataContext();
            int clienteExistente = bd.sp_usuarioExiste(this.UserNameTextBox.Text);
            bd.Log = Console.Out;
            if (clienteExistente > 0) {
                this.ErrorLabel.Text = "El usuario ya existe, pruebe con otro nombre";

            } else {
                SHA512Managed encriptar = new SHA512Managed();
                byte[] pass = encriptar.ComputeHash(
                                        Encoding.UTF8.GetBytes(this.PasswordTextBox.Text)
                                                    );
                string password = Convert.ToBase64String(pass);

                Cliente c = new Cliente();
                c.NombreUsuario = this.UserNameTextBox.Text;
                c.PasswordUsuario = password;
                c.Nombre = this.NombreTextBox.Text;
                c.Apellidos = this.ApellidosTextBox.Text;
                c.Calle = this.CalleTextBox.Text;
                c.Ciudad = this.CiudadTextBox.Text;
                c.Estado = this.EstadoTextBox.Text;
                c.Telefono = this.TelefonoTextBox.Text;
                c.Email = this.EmailTextBox.Text;
                c.FechaRegistro = DateTime.Now;
                if (this.PersonaRadioButton.Checked == true) {
                    this.ErrorLabel.Text = this.PersonaRadioButton.Text;
                    c.IDTipoCliente = 1;//Persona
                } else if (this.AsociacionRadioButton.Checked == true) {
                    this.ErrorLabel.Text = this.AsociacionRadioButton.Text;
                    this.ApellidosRequiredFieldValidator.IsValid = true;
                    c.IDTipoCliente = 2; //Asociacion
                }
                bd.Clientes.InsertOnSubmit(c);
                bd.SubmitChanges();
                bd.Dispose();
                Response.Redirect("registroUsuario.aspx");
            }
        }
        catch(SqlException) {
            this.ErrorLabel.Text = "Ocurrio un error al registrar el cliente, intentelo más
tarde";
        }
    }
}
```

Como se puede observar, primero se evalúan que las contraseñas sean las mismas, posteriormente se procede a crear una instancia de un objeto *DataContext* para poder acceder al procedimiento almacenado *sp_usuarioExiste* a través del método del mismo nombre el cual nos sirve para saber si existe un usuario con el mismo nombre, si es así se muestra un mensaje de texto indicando que el usuario ya existe, en caso contrario se procede a encriptar la contraseña que introdujo el usuario a través de la clase de algoritmo de cifrado *SHA512Managed*. Esta clase calcula el algoritmo hash (dispersión) SHA512 de la cadena de entrada y cuyo tamaño de este valor hash es de 512 bits. El método encargado de realizar la encriptación es el método *ComputeHash* el cual nos regresará un arreglo de bytes el cuál será pasado a una cadena para poder agregársela a un objeto *Cliente*. Finalmente

se crea un objeto Cliente y se le pasan los valores en los campos para que sea agregado a la base de datos.

El procedimiento almacenado mencionado anteriormente está definido dentro de SQL Server en la base de datos *ArteOnLine* de la siguiente forma:

```
CREATE procedure sp_usuarioExiste(@NombreUsuario nvarchar(30))
AS
BEGIN
Declare @NumeroRegistros int;
SELECT @NumeroRegistros=COUNT(*)
       FROM Cliente WHERE Cliente.NombreUsuario=@NombreUsuario;

return @NumeroRegistros;
END
```

El método en .NET resultado del mapeo es el siguiente:

```
[Function(Name="dbo.sp_usuarioExiste")]
public int sp_usuarioExiste(
               [Parameter(Name="NombreUsuario", DbType="NVarChar(30)")]
               string nombreUsuario) {
       IExecuteResult result = this.ExecuteMethodCall( this,
                                              ((MethodInfo)(MethodInfo.GetCurrentMethod())),
                                               nombreUsuario);
       return ((int)(result.ReturnValue));
}
```

5.5.3.6 Login de Usuario.

En este formulario se realiza el login del usuario a través de un nombre y contraseñas proporcionadas durante el proceso de registro, con esto se tendrá acceso al carrito de compra, al agregado de elementos en el mismo y al seguimiento de las órdenes que haya realizado en ciertas ocasiones. El Código asociado a la página es la siguiente:

```
<!-- tabla exterior!-->
<table style="border-style: solid; border-color: #000000; width: 400px; height: 200px;
    background-color: #B9AF6F; margin-left: auto; margin-right: auto;" align="center">
    <tr>
        <td>
            <!-- tabla interior!-->
            <table style="width: 350px; background-color: #B9AF6F;" align="center">
                <!-- nombre de usuario !-->
                <tr style="text-align: center">
                    <td> Username:</td>
                    <td>
                        <asp:TextBox ID="UserNameTextBox" runat="server"
                                     Width="144px">antonio</asp:TextBox>
                    </td>
                </tr>
                <!-- password !-->
                <tr style="text-align: center">
                    <td> Password: </td>
                    <td>
                        <asp:TextBox ID="PasswordTextBox" runat="server"
                                TextMode="Password" Width="142px">as207900</asp:TextBox>
                    </td>
                </tr>
                <!-- boton para login!-->
                <tr>
                    <td colspan="2" align="center">
                        <asp:Label ID="ErrorLabel" runat="server"
                                   Font-Names="Calibri" ForeColor="Red"></asp:Label>
                        <br />
                        <asp:Button ID="EntrarButton" runat="server"
                                  Style="text-align: center;" Text="Entrar"
                                  OnClick="EntrarButton_Click" />
```

```
                    </td>
                </tr>
            </table>
        </td>
    </tr>
</table>
```

En esta parte del código solo se muestra el código asociado a la definición en particular de la página de *login*, ya que el anterior código es el mismo. Como se observa se define una tabla exterior que contiene a una tabla interior y dentro de esta tabla los elementos que servirán para que el cliente coloque el nombre de usuario y password.

El código asociado a esta página (*code-behind)* es la siguiente:

```
protected void EntrarButton_Click(object sender, EventArgs e)
{
    try
    {
        if (string.IsNullOrEmpty(this.PasswordTextBox.Text) ||
           string.IsNullOrEmpty(this.UserNameTextBox.Text))
        {
            this.ErrorLabel.Text = "Nombre o password de usuario vacíos";
        }
        else {
            SHA512Managed encriptar = new SHA512Managed();
            byte[] pass =
                     encriptar.ComputeHash(Encoding.UTF8.GetBytes(this.PasswordTextBox.Text));
            string password = Convert.ToBase64String(pass);
            ArteOnlineDataContext bd = new ArteOnlineDataContext();
            var query = from cliente in bd.Clientes
                        where cliente.NombreUsuario.Equals(this.UserNameTextBox.Text)
                        && cliente.PasswordUsuario.Equals(password)
                        select cliente;

            if (query.Count() > 0) {
                   FormsAuthentication.RedirectFromLoginPage(
                                query.Single().NombreUsuario+","+
                                query.Single().IDCliente, false);
            } else {
                this.ErrorLabel.Text = "Nombre de usuario o password incorrectos";
            }
        }
    } catch (Exception){
        this.ErrorLabel.Text = "Nombre de usuario o password incorrectos";
    }
}
```

Como se observa se valida que los campos no estén vacíos, posteriormente se encripta la contraseña proporcionada por el usuario a través del método de instancia *ComputeHash* de la clase *SHA512Managed*, el resultado del valor *hash* de tamaño de 512 bits es convertido a una cadena y el nombre de usuario son usados como criterio de selección en el momento de la extracción del cliente de la base de datos a través de una instancia de *DataContext*, si existe un cliente con el nombre y contraseña se autentica al usuario a través de la clase *FormsAuthentication*. Con el método RedirectFromLoginPage se redirige un usuario autenticado a la dirección *URL* protegida definida en la propiedad *DefaultUrl* (dirección predeterminada *default.aspx*), se le pasa el nombre del usuario y se especifica que no se creará una cookie persistente (*false*).

5.5.3.7 Muestra de artesanías (Default.aspx).

En la página default.aspx se muestran las artesanías de la base de datos junto con su correspondiente imagen, además de que se pueden agregar elementos al carrito de compra para poder realizar una orden, siempre y cuando el usuario este logeado, ya que si no lo está se redirigirá hacia una página indicando que no se ha autenticado ente el sistema.

```
<asp:GridView runat="server" ID="artesaniasGridView" CssClass="GridViewStyle" RowStyle-
CssClass="GridViewRowStyle"
    PagerStyle-CssClass="GridViewPagerStyle" AlternatingRowStyle-
CssClass="GridViewAlternatingRowStyle"
    HeaderStyle-CssClass="GridViewHeaderStyle" AllowPaging="True"
OnPageIndexChanging="ProductosGridView_PageIndexChanging"
    AutoGenerateColumns="False" GridLines="None" OnRowCommand="ProductosGridView_RowCommand"
    HorizontalAlign="Center" PageSize="8">
    <Columns>
        <asp:TemplateField HeaderText="Imagen" ItemStyle-Width="100px">
            <ItemTemplate>
                <asp:Image ID="Image1" runat="server" ImageUrl='<%#
"FotoHandler.ashx?IDArtesania=" + Eval("IDArtesania") %>' />
            </ItemTemplate>
            <ItemStyle Width="100px"></ItemStyle>
        </asp:TemplateField>

        <!-- asp:BoundField DescripciónArtesania !-->
        <!-- asp:BoundField Cantidad !-->
        <!-- asp:BoundField Precio !-->
        <!-- asp:BoundField Artesano !-->

        <asp:TemplateField HeaderText="Carrito" ItemStyle-Width="100px"
                           ItemStyle-HorizontalAlign="Center">
            <ItemTemplate>
                <asp:LinkButton ID="AgregarLinkButton" runat="server"
                        CommandName="Agregar" CommandArgument='<%# Eval("IDArtesania") %>'
                    Text="Agregar" />
            </ItemTemplate>
            <ItemStyle HorizontalAlign="Center" Width="100px"></ItemStyle>
        </asp:TemplateField>
    </Columns>
    <RowStyle CssClass="GridViewRowStyle" Font-Size="Medium"></RowStyle>
    <PagerStyle CssClass="GridViewPagerStyle"></PagerStyle>
    <HeaderStyle CssClass="GridViewHeaderStyle"></HeaderStyle>
    <AlternatingRowStyle CssClass="GridViewAlternatingRowStyle"></AlternatingRowStyle>
</asp:GridView>
```

Como se puede observar, se define un *GridView* de *ASP.NET* sobre el cual se mostrarán todas las artesanías de la base de datos. Se tiene, que en el elemento *CssClass* se define la clase css a la que pertenece: RowStyle-CssClass y *AlternatingRowStyle-CssClass* se definen los aspectos de los renglones alternados en donde se define el color de fondo para los renglones alternados, tipo, color y tamaño de la fuente, estilo para el *anchor* (*a href*) definido en las columnas dentro del *GridView*, definición del *anchor* al pasar el puntero sobre este; *HeaderStyle-CssClass* para especificar la cabecera del *GridView*, el tamaño en pixeles, el color, tipo, y tamaño de la fuente para la letra; *AllowPaging* para permitir que se puede paginar sobre el mismo *GridView* . A demás de lo anterior, se agregan columnas, dentro de estas se agregan elementos de *ASP.NET* junto con elementos *html* y se da una asignación a los valores *DataField* que tendrán un enlace de datos con la fuente *DataSource* en el código *C#,* y estas deben coincidir con las propiedades de las clase a la cual se están enlazando.

5.5.3.7.1 Archivo FotoHandler.ashx para el manejo de imágenes.

Cabe mencionar que en uno de los elementos se hace uso de la siguiente expresión:

```
<Columns>
        <asp:TemplateField HeaderText="Imagen" ItemStyle-Width="100px">
            <ItemTemplate>
                <asp:Image ID="Image1" runat="server"
                 ImageUrl='<%# "FotoHandler.ashx?IDArtesania=" + Eval("IDArtesania") %>' />
            </ItemTemplate>
<!—Definición de los demás elementos dentro del GridView !-->
```

El archivo *FotoHandler.ashx* es un controlador *HTTP* (*HttpHandler)*, específicamente un controlador *HTTP* web genérico *.ashx*, el cuál procesa las peticiones HTTP en una aplicación web asignadas por *ASP.NET*. Dado que es genérico, no contiene ninguna interfaz para mostrar al usuario, por lo que simplemente procesa las solicitudes y devuelve los resultados.

La función básica de este archivo es la de poder trabajar con las imágenes provenientes de la base de datos, asignarlas al elemento *asp:Image* en el *GridView*. La expresión que se usa para efectuar lo anterior es: *"FotoHandler.ashx?IDArtesania=" + Eval("IDArtesania") %>'*, donde la propiedad *IDArtesania* es la encargada de buscar la imagen correcta a insertar, el cuál es asociado cuando se enlazan los datos con la propiedad *Datasource* del *GridView*. La definición del código *C#* (*code-behind)* que procesa la petición es la siguiente:

```
public class FotoHandler : IHttpHandler, IReadOnlySessionState {
    private List<Artesania> artesanias = new List<Artesania>();
    /*
     * Cuando se necesita un controlador HTTP, ASP.NET llama al método ProcessRequest en el
       controlador adecuado.
     * Los controladores HTTP tienen acceso al contexto de la aplicación
     */
    public void ProcessRequest(HttpContext context) {
        int IDArtesania = Convert.ToInt32(context.Request.QueryString["IDArtesania"]);
        artesanias = new ArteOnlineDataContext().Artesanias.ToList();
        Artesania ar = artesanias.Find(p => p.IDArtesania == IDArtesania);
        //El método ProcessRequest del controlador crea una respuesta, que
        //se devuelve al explorador que realizó la solicitud.
        //Se especifíca el tipo MIME que tendrá el stream sobre la respuesta
        //para que el navegador lo abra correctamente
        context.Response.ContentType = "image/jpeg";
        //Se escriben los valores binarios sobre el stream de la respuesta
        context.Response.BinaryWrite(ar.Imagen.ToArray());
    }
    public bool IsReusable { get { return false; } }
}
```

Como se observa, para poder crear el *HTTPHandler*, la clase *FotoHandler* implementa la interfaz *IHttpHandler* con lo que se especifica que es síncrono y la interfaz *IReadOnlySessionState* para especificar que solo se accede al estado de la sesión en solo lectura (esta interfaz no tiene métodos). La interfaz *IHttpHandler* requiere que se implemente la propiedad *IsReusable* y el método *ProcessRequest*.

El método *ProcessRequest*, recibe como único parámetro un *HttpContext*, el cual representa el contexto de la aplicación, de esta forma cuando se llama al controlador, ASP.NET llama al método *ProcessRequest* del controlador adecuado (*FotoHandler.ashx*). El código que escribe en el método *ProcessRequest* del controlador crea una respuesta, que se devuelve la solicitud. En esta caso se obtiene el *IDArtesania* correspondiente al valor encontrado en el método *QueryString["IDArtesania"],* este valor (*IDArtesania*) es el que se colocó cuando se llamó al controlador y corresponde a uno de los campos existentes en el tipo de los datos cargados desde la base de datos (se cargan objetos del tipo Artesania) que se encuentran en la lista genérica *artesanias*. Posteriormente a través de este valor, se localiza la artesanía y su correspondiente imagen, para posteriormente devolver los valores de la imagen sobre la respuesta y mostrarla sobre el *GridView* del formulario *default.aspx*.

La propiedad *IsReusable* se usa para especificar si se puede colocar el controlador en un grupo y reutilizarlo para aumentar el rendimiento. Por otra parte el código (*code-behind)* que se encarga de la carga de los datos desde la base de datos es el siguiente:

```
//Definición de los campos a usar
private List<Orden> listaOrdenes = new List<Orden>();
private string[] userIdentity = new string[2];
private ShoppingCart instanciaShoppingCart = null;

protected void Page_Load(object sender, EventArgs e){
    try
```

```
    {
        userIdentity = User.Identity.Name.Split(new char[] { ',' });
        this.BienvenidoLabel.Text = (String.IsNullOrEmpty(User.Identity.Name)) ?
            "Usuario: Anónimo" : "Usuario: " + userIdentity[0];

        if (HttpContext.Current.Session["Busqueda"] != null) {
            List<ArtesaniasJoin> artesanias = new List<ArtesaniasJoin>();
            artesanias = (List<ArtesaniasJoin>)HttpContext.Current.Session["Busqueda"];

            this.artesaniasGridView.DataSource = artesanias;
            this.artesaniasGridView.DataBind();
        } else {
            ArteOnlineDataContext bd = new ArteOnlineDataContext();
            var query = from artesanias in bd.Artesanias
                        join artesanos in bd.Artesanos
                        on artesanias.IDArtesano equals artesanos.IDArtesano
                        select new ArtesaniasJoin
                        {
                            IDArtesania = artesanias.IDArtesania,
                            DescripcionArtesania = artesanias.DescripcionArtesania,
                            Cantidad = artesanias.Cantidad,
                            Precio = artesanias.Precio,
                            NombreArtesano = artesanos.Nombre + " " + artesanos.Apellidos
                        };

            this.artesaniasGridView.DataSource = query.ToList();
            this.artesaniasGridView.DataBind();
        }
    } catch (Exception){     }
}
```

En primera instancia se verifica que exista un usuario que haya iniciado sesión, si es así se muestra el nombre en caso contrario solo se muestra que es anónimo. Posteriormente se verifica que en la sesión *Busqueda* existan valores, si es así quiere decir que previamente se cargaron datos resultado de una búsqueda, si no lo es, entonces se crea una instancia de *DataContext* y se extraen los datos necesarios de las artesanías y los resultados son asignados al *GridView* a través de la propiedad *DataSource* y finalmente se llama al método *DataBind* para realizar el enlace de datos y mostrarlos sobre el *GridView*.

5.5.3.8 Búsqueda de Artesanías.

Esta opción existe en cada una de los formularios *.aspx* de la aplicación web *ArteOnLine*, por lo que el código y proceso que se muestra a continuación es el mismo en todas. Para este proceso se usa como criterio principal el nombre de la artesanía y cuyo acceso se puede obtener mediante un campo de texto vacío y un botón. El código (*code-behind)* asociado es el siguiente:

```
protected void BuscarButton_Click(object sender, EventArgs e){
    ArteOnlineDataContext bd = new ArteOnlineDataContext();
    var query = from artesanias in bd.Artesanias
                join artesanos in bd.Artesanos
                on artesanias.IDArtesano equals artesanos.IDArtesano
                where artesanias.DescripcionArtesania.Contains(this.BuscarTextBox.Text)
                select new ArtesaniasJoin {
                    IDArtesania = artesanias.IDArtesania,
                    DescripcionArtesania = artesanias.DescripcionArtesania,
                    Cantidad = artesanias.Cantidad,
                    Precio = artesanias.Precio,
                    NombreArtesano = artesanos.Nombre + " " + artesanos.Apellidos
                };
    HttpContext.Current.Session["Busqueda"] = query.ToList();
    bd.Dispose();
    Response.Redirect("Default.aspx");
}
```

Como se ve, el proceso es simple, se extraen los datos de las artesanías a través de una instancia de *DataContext* y los resultados son pasados a una sesión llamada *Busqueda*, para finalmente direccionar hacia la página *Default.aspx*, en la cual se mostrarán las artesanías con el proceso mostrado en la sección *Página Default (muestra de artesanías)*.

5.5.3.9 Agregar Ítems al carrito.

5.5.3.9.1 Default.aspx.

La ejecución de este proceso localizado dentro del formulario *Default.aspx*, reside específicamente en el elemento *asp:LinkButton* de la última columna del *GridView*. Para comprenderlo mejor se expone la parte del código que tiene asignada la función:

```
<asp:TemplateField HeaderText="Carrito" ItemStyle-Width="100px"
                   ItemStyle-HorizontalAlign="Center">
       <ItemTemplate>
              <asp:LinkButton ID="AgregarLinkButton" runat="server"
                              CommandName="Agregar"
                              CommandArgument='<%# Eval("IDArtesania") %>'
       </ItemTemplate>
       <ItemStyle HorizontalAlign="Center" Width="100px"></ItemStyle>
</asp:TemplateField>
```

Cuando se presiona el elemento *asp:LinkButton* se ejecuta una acción y se le asignan dos elementos importantes, El primero corresponde al valor de *CommandName* es “Agregar” y servirá para saber que comando se está presionando y el segundo corresponde al parámetro que se le pasa al comando, este valor, *IDArtesania*, se encuentra en una expresión de enlace de datos (contenidas dentro de los delimitadores <%# y %>) y se utiliza el método *Eval*. El código *C#* (*code-behind)* que controla la operación de agregado es el siguiente:

```
//campo usado
private ShoppingCart instanciaShoppingCart = null;

protected void ProductosGridView_RowCommand(object sender, GridViewCommandEventArgs e)
{
    if (e.CommandName == "Agregar"){
        if (String.IsNullOrEmpty(User.Identity.Name)){
            Response.Redirect("sinLogin.aspx");
        } else {

            //Se obtiene el IDArtesania del GridView (através del elemento
            //e.CommandArgument) asociado al asp:LinkButton del asp:ItemTemplate del
            //asp:TemplateField
            instanciaShoppingCart = new ShoppingCart();
            instanciaShoppingCart.AgregarItem(Convert.ToInt32(e.CommandArgument));
            HttpContext.Current.Session["ArteOnLineShopingCart"] =
                                        instanciaShoppingCart.Instance;
        }
    }
}
```

Como se observa, se obtiene el valor del *CommandArgument* del *GridView* y se evalúa si el nombre es “Agregar”, posteriormente se verifica si el usuario se ha logeado, posteriormente se realiza el agregado del ítem hacia el carrito de compra a través de una instancia de la clase *ShoppingCart* y del método *AgregarItem*. Esté método recibe como parámetro el id de la artesanía y posteriormente se agrega la instancia de *ShoppingCart* a la sesión *ArteOnLineShoppingCart*.

5.5.3.9.2 Ensamblado ShoppingCart.dll.

Se ha mencionado la clase *ShoppingCart* así que ahora toca explicar de forma resumida cuál es la función de esta clase. Esta clase reside en el ensamblado *ShopingCart.dll*, el cuál fue

creado y compilado por separado. Este ensamblado contiene las siguientes clases, ordenadas por nivel de operación:

1. ShoppingCart.cs.
2. CartItem.cs.
3. Productos.cs.

5.5.3.9.2.1 ShoppingCart.

Esta clase es la que se expone sobre la aplicación web, y es la que se utiliza de forma directa para poder realizar las operaciones de agregado, modificación de cantidad y remoción de elementos sobre el carrito de compra, todo esto se realiza en una lista genérica de tipo *CartItem*. El código es el siguiente:

```
public class ShoppingCart
{
    //Campos
    private List<CartItem> items;
    private ShoppingCart instance;

    //Propiedades
    public List<CartItem> Items {get { return this.items; } }

    public ShoppingCart Instance {
     get
        {
            if (HttpContext.Current.Session["ArteOnLineShoppingCart"] == null) {
                instance = new ShoppingCart();
                instance.items = new List<CartItem>();
                HttpContext.Current.Session["ArteOnLineShoppingCart"] = instance;
            } else{
                instance =
                (ShoppingCart)HttpContext.Current.Session["ArteOnLineShoppingCart"];
            }
            return instance;
        }
    }

    //constructor
    public ShoppingCart(){ }

    //Métodos para modificación de Item
    public void AgregarItem(int artesaniaId) {
        CartItem newItem = new CartItem(artesaniaId);
        if (Instance.items.Contains(newItem)){
            foreach (CartItem item in Instance.items){
                if (item.Equals(newItem)){
                    item.Cantidad++;
                    return;
                }
            }
        } else {
            newItem.Cantidad = 1;
            Instance.items.Add(newItem);
        }
    }

    public void ColocarCantidad(int artesaniaId, int cantidad){
        if (cantidad == 0) {
            RemoverItem(artesaniaId);
            return;
        }

        CartItem updatedItem = new CartItem(artesaniaId);
        foreach (CartItem item in Instance.items) {
            if (item.Equals(updatedItem)) {
                item.Cantidad = cantidad;
                return;
```

```
            }
        }
    }

    public void RemoverItem(int artesaniaId){
        CartItem removedItem = new CartItem(artesaniaId);
        Instance.items.Remove(removedItem);
    }

    //Métodos de reporte
    public decimal GetSubTotal(){
        decimal subTotal = 0;
        foreach (CartItem item in Instance.items)
            subTotal += item.Total;
        return subTotal;
    }

    public decimal GetTotal() {
        decimal subTotal = 0M;
        decimal total = 0M;
        foreach (CartItem item in Instance.items)
            subTotal += item.Total;
        return total;
    }
}
```

5.5.3.9.2.2 CartItem.

Esta clase es la intermediaria entra la clase Shoppingcart y la clase Productos. Esta clase es la encargada de pasar los datos o la información de las artesanías provenientes de la clase Productos hacia la clase *Shoppingcart* a través de campos que correspondan con el tipo de campo que se extrae de la base de datos. Esta clase implementa la interfaz *IEquatable<T>* donde *T* es *CartItem* y cuyo miembro que implementa es el método *Equals*, con este método se determina la igualdad de instancias entre los productos a través de la propiedad *IDArtesania*. El código que implementa esta lógica es el siguiente:

```
public class CartItem : IEquatable<CartItem>
{
    #region campos
    private int ArtesaniaId;
    private Productos producto = null;
    #endregion

    #region Constructor
    public CartItem(int artesaniaId){
        this.ArtesaniaId = artesaniaId;
    }
    #endregion

    #region IEquatable<CartItem> Members
    public bool Equals(CartItem item){
        return item.IDArtesania == this.ArtesaniaId;
    }
    #endregion

    #region Propiedades

    public Productos Prod
    {get
        {if (producto == null)
            {producto = new Productos(this.ArtesaniaId);}
            return producto;
        }
    }

    public int IDArtesania
    {get { return ArtesaniaId; }
     set {producto = null;
          ArtesaniaId = value;
```

```
        }
    }

    public int IDArtesano{get { return Prod.IDArtesano; }}
    public String DescripcionArtesania {get { return Prod.DescripcionArtesania; }}
    public int Cantidad{get { return Prod.Cantidad;} set { Prod.Cantidad = value; }}
    public int Cantidades {get { return Prod.Cantidades; }}
    public decimal Precio {get { return Prod.Precio; }}
    public int IDClasificacion{get { return Prod.IDClasificacion; }}
    public Binary Imagen {get { return Prod.Imagen; }}
    public decimal SubTotal{get { return Precio * Cantidad; }
    }
    #endregion
}
```

5.5.3.9.2.3 Productos.

Esta clase trabaja en el nivel inferior y es la encargada de obtener un número determinado de campos de la artesanía desde la base de datos *ArteOnLine* de acuerdo al id de la artesanía que le sea pasado por la clase *CartItem* al momento de ser instanciada. Las propiedades con las que cuenta esta clase son del mismo tipo y nombre que la de la clase *CartItem* excepto por el campo *Cantidad*. El código de la lógica mencionada es la siguiente:

```
public class Productos{
    //Campos
    private Artesania producto = new Artesania();

    //Propiedades
    public int IDArtesania { get; set; }
    public int IDArtesano { get; set; }
    public String DescripcionArtesania { get; set; }
    public int Cantidades { get; set; }
    public int Cantidad { get; set; }
    public decimal Precio { get; set; }
    public int IDClasificacion { get; set; }
    public Binary Imagen { get; set; }

    // Constructor
    public Productos(int productoId) {
        ArteOnlineDataContext bd = new ArteOnlineDataContext();
        producto = (from p in bd.Artesanias
                    where p.IDArtesania==productoId select p).Single();
        bd.Dispose();
        if (producto != null) {
            this.IDArtesania = producto.IDArtesania;
            this.IDArtesano = producto.IDArtesano;
            this.IDClasificacion = producto.IDClasificacion;
            this.DescripcionArtesania = producto.DescripcionArtesania;
            this.Cantidades = producto.Cantidad;
            this.Precio = producto.Precio;
            this.Imagen = producto.Imagen;
        }
    }
}
```

5.5.3.9.3 Formulario Carrito.aspx.

Una vez que se agregaron los elementos al carrito de compra, se puede ir al formulario carrito.aspx para poder observar las artesanías que se agregaron y poder modificar la cantidad o eliminarlos. Este formulario tiene definido un *GridView* para poder mostrar las artesanías, en donde también se observan los datos como: imagen, descripción de la artesanía, cantidad que se piden, precio, total y las opciones de eliminar ó modificar. En la definición del *GridView* se utiliza la misma lógica usada en el formulario Default.aspx, por lo que solo se mostrará el código de marcado centrado en la definición de las columnas dentro del GridView:

```
<Columns>
    <!-- Imagen !-->
    <asp:TemplateField HeaderText="Imagen" ItemStyle-Width="100px">
        <ItemTemplate>
            <asp:Image ID="Image1" runat="server"
                ImageUrl='<%# "../FotoHandler.ashx?IDArtesania=" + Eval("IDArtesania") %>' />
        </ItemTemplate>
        <ItemStyle Width="100px"></ItemStyle>
    </asp:TemplateField>
    <!-- Descripcion de la artesanía!-->
    <asp:BoundField HeaderText="Descripción" DataField="DescripcionArtesania"
                    ItemStyle-Width="200">
        <ItemStyle Width="200px"></ItemStyle>
    </asp:BoundField>
    <!-- cantidades de artesanías a pedir!-->
    <asp:TemplateField HeaderText="Cantidad" ItemStyle-Width="100px">
        <ItemTemplate>
            <asp:TextBox runat="server" ID="CantidadTextBox" Columns="5"
                         Text='<%# Eval("Cantidad") %>'
                Width="30px"></asp:TextBox>
            <asp:Label runat="server" ID="Label1" Columns="5" Text="de:"></asp:Label>
            <asp:Label runat="server" ID="CantidadesLabel" Columns="5"
                       Text='<%# Eval("Cantidades") %>'></asp:Label>
            <!--'<%# Eval("Cantidades") %>' -->
        </ItemTemplate>
        <ItemStyle Width="100px"></ItemStyle>
    </asp:TemplateField>
    <!-- Precio a pagar por las artesanías del mismo tipo!-->
    <asp:BoundField HeaderText="Precio" DataField="Precio" ItemStyle-Width="100"
                    DataFormatString="{0:C}">
        <ItemStyle Width="100px"></ItemStyle>
    </asp:BoundField>
    <!-- Precio a pagar por las artesanías totales!-->
    <asp:BoundField HeaderText="Total" DataField="Total" ItemStyle-Width="100"
                    DataFormatString="{0:C}">
        <ItemStyle Width="100px"></ItemStyle>
    </asp:BoundField>
    <!-- opción de eliminacion!-->
    <asp:TemplateField HeaderText="Eliminar" ItemStyle-Width="100px"
                       ItemStyle-HorizontalAlign="Center">
        <ItemTemplate>
            <asp:LinkButton ID="EliminarLinkButton" runat="server"
                            CommandName="Eliminar"
                            CommandArgument='<%# Eval("IDArtesania") %>'
                            Text="Eliminar" />
        </ItemTemplate>
        <ItemStyle HorizontalAlign="Center" Width="100px"></ItemStyle>
    </asp:TemplateField>
    <!-- opción de modificación !-->
    <asp:TemplateField HeaderText="Modificar" ItemStyle-Width="100px"
                       ItemStyle-HorizontalAlign="Center">
        <ItemTemplate>
            <asp:LinkButton ID="ModificarLinkButton" runat="server"
                             CommandName="Modificar"
                             CommandArgument='<%# Eval("IDArtesania") %>'
                Text="Modificar" OnClick="ModificarLinkButton_Click" />
        </ItemTemplate>
        <ItemStyle HorizontalAlign="Center" Width="100px"></ItemStyle>
    </asp:TemplateField>
</Columns>
```

Por su parte, el código C# (*code-behind*) encargado de mostrar los datos cuando se carga el formulario carrito.aspx es el siguiente:

```
protected void Page_Load(object sender, EventArgs e)
{
    userIdentity = User.Identity.Name.Split(new char[] { ',' });
    this.BienvenidoLabel.Text = "Usuario: " + userIdentity[0];

    instanciaShoppingCart = new ShoppingCart();
```

```
    //Para no estar reemplazando los valores que tenemos en el DataGidView cada vez
    //que la pagina se carga al presionar cualquier boton, por que si no, todos los
    //valores que agregemos o coloquemos en el textbox de cantidad, sera reemplazado
    //por los antiguos valores de los objetos en listaProductos
    if (!IsPostBack){
        if (listaProductos.Count < 1){
            this.FinalizarCarritoButton.Enabled = false;
        } else {
            this.FinalizarCarritoButton.Enabled = true;
            this.artesaniasGridView.DataSource = instanciaShoppingCart.Instance.Items;
            this.artesaniasGridView.DataBind();
        }
    }
}
```

En el código mostrado anteriormente se obtiene la identidad del usuario, posteriormente se crea una instancia de la clase *ShoppingCart* mencionada anteriormente y a través de la propiedad *Instance* se obtienen los elementos que contiene *Items* y son asignados al *GridView* a través de la propiedad *DataSource* y se realiza el enlace de datos para mostrarlos sobre el *GridView*.

5.5.3.10 Eliminar ítem del carrito.

Para pode eliminar objetos en el formulario carrito.aspx se utiliza el siguiente fragmento de código de marcado definido en uno de los elementos del *GridView* para poder ejecutar la acción:

```
<asp:LinkButton ID="EliminarLinkButton" runat="server" CommandName="Eliminar"
                CommandArgument='<%# Eval("IDArtesania") %>'
                Text="Eliminar" />
```

El código asociado (*code-behind*) para manipular la modificación de la cantidad del elemento en el carrito es el siguiente:

```
protected void ProductosGridView_RowCommand(object sender, GridViewCommandEventArgs e) {
    if (e.CommandName == "Eliminar") {
        instanciaShoppingCart.Instance.RemoverItem(Convert.ToInt32(e.CommandArgument));
        this.artesaniasGridView.DataSource = instanciaShoppingCart.Instance.Items;
        this.artesaniasGridView.DataBind();
    }
}
```

La lógica implementada es similar a la vista en la sección Agregar ítems al carrito, se evalúa la propiedad *CommandName* para saber si se presionó la opción "*Eliminar*" a través del nombre asignado en la definición del *GridView*, si es así se utiliza de nuevo la instancia de *ShoppingCart* y con el método *RemoveItem* se elimina la artesanía de acuerdo al id de la artesanía, posteriormente se reasigna la fuente de datos y se enlazan con el *GridView*.

5.5.3.11 Modificar ítem del carrito.

Para pode modificar la cantidad de objetos que se solicitan en el formulario *carrito.aspx* se utiliza el siguiente fragmento de código de marcado definido en uno de los elementos del *GridView* para poder ejecutar la acción:

```
<asp:LinkButton ID="ModificarLinkButton" runat="server" CommandName="Modificar"
                CommandArgument='<%# Eval("IDArtesania") %>'
                Text="Modificar" OnClick="ModificarLinkButton_Click" />
```

El código asociado para la modificación de la cantidad (*code-behind*) es el siguiente:

```
protected void ModificarLinkButton_Click(object sender, EventArgs e) {
    foreach (GridViewRow renglon in this.artesaniasGridView.Rows){
        //serciorarnos de que se esta va a seleccionar un DataRow
        if (renglon.RowType == DataControlRowType.DataRow) {
            try {
                int indiceRenglon = renglon.RowIndex;
                // Obtener el IDAtresania desde el datakeys del GridView
                int idArtesania =
                      Convert.ToInt32(this.artesaniasGridView.DataKeys[indiceRenglon].Value);
                // Encontrar el TextBox de la Cantidad y recibir el valor
                int cantidad =
                  int.Parse(((TextBox)renglon.Cells[2].FindControl("CantidadTextBox")).Text);

                int cantidades =
                  int.Parse(((Label)renglon.Cells[2].FindControl("CantidadesLabel")).Text);
                if (cantidad > cantidades || cantidad < 0){
                    this.EstadoLabel.Text = "Una de las cantidades es superior a las que
                                    están en la base de datos o menor a cero.Verifice la
                                    cantidad";
                    return;
                } else{
                    instanciaShoppingCart.Instance.ColocarCantidad(idArtesania, cantidad);
                }
            }catch (FormatException) { }
        }
    }
    this.artesaniasGridView.DataSource = instanciaShoppingCart.Instance.Items;
    this.artesaniasGridView.DataBind();
}
```

En el código mostrado anteriormente se puede observar lo que se realiza gracias a los comentarios. Para empezar se realiza una iteración sobre cada objeto *GridViewRow* del *GridView*, con esto en cada iteración se verifica que el tipo del *GridViewRow* sea un *DataRow*. Posteriormente se obtiene el índice del renglón y el id de la artesanía usando el índice del renglón a través del valor *value* de la propiedad *DataKeys* asociado al *GridView*, como se observa en el siguiente código que es parte de la definición del *GridView* en el formulario *carrito.aspx*:

```
<asp:GridView ID="artesaniasGridView" runat="server" CssClass="GridViewStyle"
      ..... otras definiciones del GridView
    DataKeyNames="IDArtesania" PageSize="8">
```

Se puede observar que la definición del *GridView* sigue la misma estructura que la mostrada en las anteriores definiciones sobre otros formularios que contienen controles *GridView*, pero destaca la propiedad *DataKeyNames* el cual tiene asociado el valor *IDArtesania*. El objeto *DataKey* contiene los valores del campo especificado en la propiedad *DataKeyNames*. Eso quiere decir que este valor contiene el *IDArtesania* de la fuente de datos a la cuál es enlazada.

Una vez obtenido el id de la artesanía se procede a buscar sobre cada celda del renglón a los controles *CantidadTextBox* y *CantidadesLabel* para asignarlos a las variables del tipo *int* cantidad y cantidades respectivamente. Finalmente los datos son de nuevo enlazados al control *GridView*.

5.5.3.12 Creación de la orden de compra.

Esta operación se efectúa dentro del formulario *carrito.aspx* dentro del método asociado al botón *FinalizarCarritoButton*. Con este método se crea la orden del cliente sobre la base de datos y se establece el estado de la orden a un nivel 1, lo que significa que la orden ha sido iniciada. El código que efectúa este proceso es el siguiente:

```
protected void FinalizarCarritoButton_Click(object sender, EventArgs e) {
    DateTime? fechaOrden = DateTime.Now;
    //Add... devuelve una nueva estructura DateTime cuyo valor es el resultado
    //del cálculo
```

```
    DateTime? fechaEntrega = DateTime.Now.AddDays(5);
    ArteOnlineDataContext bd = new ArteOnlineDataContext();
    bd.Connection.Open();
    bd.Transaction = bd.Connection.BeginTransaction();
    try {

        Int32? error = -1;
        Int32? idDetallePedido = -1;
        Int32? idCliente = int.Parse(this.userIdentity[1]);
        int i = 0;
        Int32? idOrden = bd.sp_CrearOrden(fechaOrden, fechaEntrega,
                                          idCliente, ref error);
        if (idOrden == 0 && error != 0) {
            bd.Transaction.Rollback();
            bd.Transaction.Dispose();
            bd.Connection.Close();
            bd.Dispose();
            this.EstadoLabel.Text = "Ocurrió un error al crear la orden";
            return;
        } else {
            foreach (CartItem item in instanciaShoppingCart.Instance.Items) {
                CrearOrden orden = new CrearOrden {
                        FechaOrden = fechaOrden,
                        FechaEntrega = fechaEntrega,
                        IDCliente = int.Parse(this.userIdentity[1]),
                        IDStatusOrden = 1,
                        IDArtesania = item.IDArtesania,
                        PrecioUnitario = item.Precio,
                        Cantidad = item.Cantidad
                    };
                idDetallePedido = bd.sp_CrearDetalleOrden(
                                            idOrden, orden.IDArtesania,
                                            orden.PrecioUnitario,
                                            orden.Cantidad, ref error);
                if (idDetallePedido == 0 && error != 0) {
                    bd.Transaction.Rollback();
                    bd.Transaction.Dispose();
                    bd.Connection.Close();
                    bd.Dispose();
                    this.EstadoLabel.Text = "Ocurrió un error al crear la orden";
                    return;
                }
                i++;
            }
            bd.Transaction.Commit();
            bd.Connection.Close();
            bd.Dispose();
            instanciaShoppingCart.Instance.Items.Clear();
            this.listaOrdenes.Clear();
            this.EstadoLabel.Text = "La orden fue creada exitosamente";
        }
    } catch (Exception ex) {
        bd.Transaction.Rollback();
        bd.Transaction.Dispose();
        bd.Connection.Close();
        bd.Dispose();
        this.EstadoLabel.Text = "*Ocurrió un error al crear la orden";
    }
    this.FinalizarCarritoButton.Enabled = false;
}
```

Esta vez se usa la propia conexión de *DataContext* y a través de esta se inicia la transacción, esto sirve para saber exactamente en qué momento realizar una operación *Commit* si la creación de la orden fue exitosa o un *Rollback* en caso contrario, esto es debido a que se usan dos métodos por separado que llaman cada uno a un procedimiento almacenado en *SQL Server*. El primer método sp_*CrearOrden*, sirve para crear la orden que contiene las fechas de creación y entrega, además del id del cliente, aunado a esto, este método recibe por referencia a un parámetro llamado *error*, el cual nos indica si existió algún error en el momento de la creación de

la orden, finalmente este método regresa el id de la orden recién creada. El procedimiento almacenado es el siguiente:

```
create procedure sp_CrearOrden(
        @fechaOrden DateTime,
        @fechaEntrega DateTime,
        @idCliente int,
        @error int OUTPUT )
As
Begin
DECLARE @IdPedido int;
INSERT INTO Orden(FechaOrden, FechaEntrega, IDCliente, IDStatusOrden)
VALUES(@fechaOrden, @fechaEntrega, @idCliente, 1);

        IF(SELECT  error=@@Error)!=0
                Begin
                        Set @IdPedido=0;
                End
        Else
                Begin
                        Set @IdPedido=@@Identity;
        End
return @IdPedido;
End
```

Para poder seguir con la creación del detalle se itera sobre todos los objetos que se encuentran en la lista genérica *Items* de la instancia de la clase *ShoppingCart* y se crean objetos *CrearOrden* a partir de los datos de los objetos en *Shoppingcart* para asignarlos a una lista genérica del mismo tipo (*CrearOrden*) y estos mismos datos nos servirán para pasarlos como parámetros en la llamada a otro método llamado sp_*CrearDetalleOrden* y que al mismo tiempo llama a un procedimiento almacenado del mismo nombre. Este procedimiento almacenado se encuentra definido de la siguiente forma en SQL Server:

```
create procedure sp_CrearDetalleOrden(
        @idPedido int,
        @idArtesania int,
        @precioUnitario Decimal(12,2),
        @cantidad int,
        @error int OUTPUT )
As
Begin
DECLARE @IdDetallePedido int;
INSERT INTO DetalleOrden(IDPedido,IDArtesania,PrecioUnitario,Cantidad)
VALUES(@IdPedido,@idArtesania,@precioUnitario,@cantidad);

        IF(SELECT error=@@Error)!=0
                Begin
                        set @IdDetallePedido=0;
                End
        Else
                Begin
                        set @IdDetallePedido=@@Identity;
                        UPDATE Artesania SET Cantidad =
                                                (SELECT Cantidad FROM Artesania
                                                Where IDArtesania=@idArtesania)-@cantidad
                        WHERE IDArtesania =@idArtesania;
        End
return @IdDetallePedido;
End
```

Como se observa, recibe como parámetros de entrada los datos necesarios para la creación del detalle de la orden los cuáles son el id de la orden a la que pertenece, el id de la artesanía, precio unitario, cantidad de artesanías que se solicitan y como parámetro de salida la variable *error* indicando que ocurrió un error al momento de realizar la creación del detalle, si es que hubo

errores, finalmente se devuelve el id de detalle de cada objeto que se inserta. Finalmente se realizan los cambios en la base de datos.

5.5.3.13 Seguimiento de las órdenes.

Una vez que se haya realizado una orden o si el cliente quiere observar los datos de las ordenes que ha realizado puede dirigirse, una vez logeado, a la página de carrito y en la opción Seguimiento de ordenes dar clic el mismo para poder observar los datos registrados de las ordenes en el formulario *seguimientoOrdenes.aspx*. En este formulario se encuentra definido un control *GridView* para poder observar los datos de la base de datos, por cuestión de espacio y como anteriormente ya se han mostrado como se define este control en secciones anteriores solo se muestra la definición de las columnas con las que cuenta este control:

```
<Columns>
    <asp:BoundField HeaderText="Orden No." DataField="IDPedido" ItemStyle-Width="30">
        <ItemStyle Width="30px"></ItemStyle>
    </asp:BoundField>
    <!-- asp:BoundField Artesania !-->
    <!-- asp:BoundField Cantidad !-->
    <!-- asp:BoundField PrecioUnitario !-->
    <!-- asp:BoundField FechaOrden !-->
    <!-- asp:BoundField FechaEntrega !-->

    <asp:BoundField HeaderText="Estado" DataField="Status" ItemStyle-Width="100">
        <ItemStyle Width="100px"></ItemStyle>
    </asp:BoundField>
</Columns>
```

Como se observa tiene columnas para el id de la orden, el nombre de la artesanía, la cantidad que se pidieron de la artesanía, el precio unitario, la fecha de la orden, la fecha de entrega y el estado en el que se encuentra la orden. Por su parte el código *C#* asociado a la carga de los datos es el siguiente:

```
protected void Page_Load(object sender, EventArgs e){
    string[] userIdentity = User.Identity.Name.Split(new char[] { ',' });
    this.BienvenidoLabel.Text = "Usuario: " + userIdentity[0];
    Int32? idClient = int.Parse(userIdentity[1]);
    ArteOnlineDataContext bd = new ArteOnlineDataContext();
    List<sp_ObtenerOrdenClienteASPNETResult> query =
                            bd.sp_ObtenerOrdenClienteASPNET(idClient).ToList();
    this.artesaniasGridView.DataSource = query;
    this.artesaniasGridView.DataBind();

}
```

Como se observa el procedimiento es simple, primero se obtiene la identidad el usuario y el id del cliente, posteriormente se crea una instancia de *DataContext* y se llama al método *sp_ObtenerOrdenClienteASPNET* para poder llamar a un procedimiento almacenado con el mismo nombre en SQL Server para poder obtener los datos del detalle de las ordenes. Este procedimiento almacenado se encuentra definido de la siguiente forma:

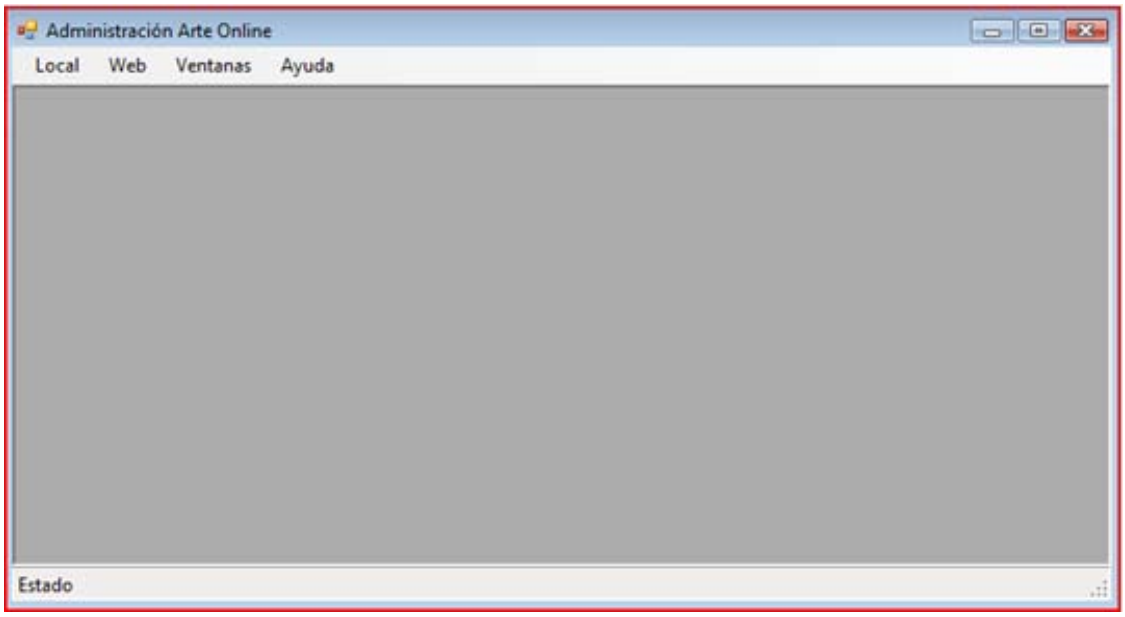
```
CREATE PROCEDURE sp_ObtenerOrdenClienteASPNET(@idCliente int)
AS
BEGIN

SELECT O.IDPedido,O.FechaOrden,O.FechaEntrega,S.Status,A.DescripcionArtesania,
       DO.Cantidad,DO.PrecioUnitario
FROM Orden As O
INNER JOIN StatusOrden As S ON O.IDStatusOrden=S.IDStatusOrden
INNER JOIN DetalleOrden As DO ON O.IDPedido=DO.IDPedido
INNER JOIN Artesania AS A ON DO.IDArtesania=A.IDArtesania
WHERE O.IDCliente=@idCliente ORDER BY O.IDPedido ASC;
END
```

Este procedimiento almacenado solo obtiene la información necesario que necesita el cliente para poder ver sus ordenes realizadas a través de operaciones *Inner Join* y usando las tablas *Orden, StatusOrden*, *DetalleOrden* y *Artesania*, esto lo realiza en base al id del cliente como parámetro de selección. El resultado es asignado a la variable *query* y esta es enlazada con el control *GridView* para finalmente mostrar los datos.

5.5.4 IMPLEMENTACIÓN DE DESPLIEGUE.

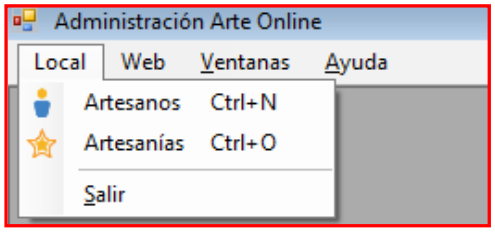
5.5.4.1 Inicio del sistema en Windows Forms.

En la siguiente imagen se muestra l ventana principal del sistema para administrar la base de datos de ArteOnLine que fue definida:

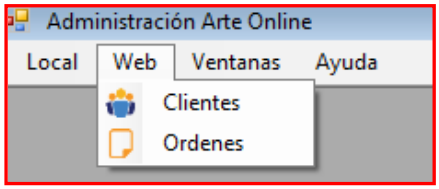


Ventana principal al iniciarse el sistema.

Las siguientes imágenes corresponden al menú para cargar las ventanas de los artesanos, artesanías, órdenes y clientes en general:



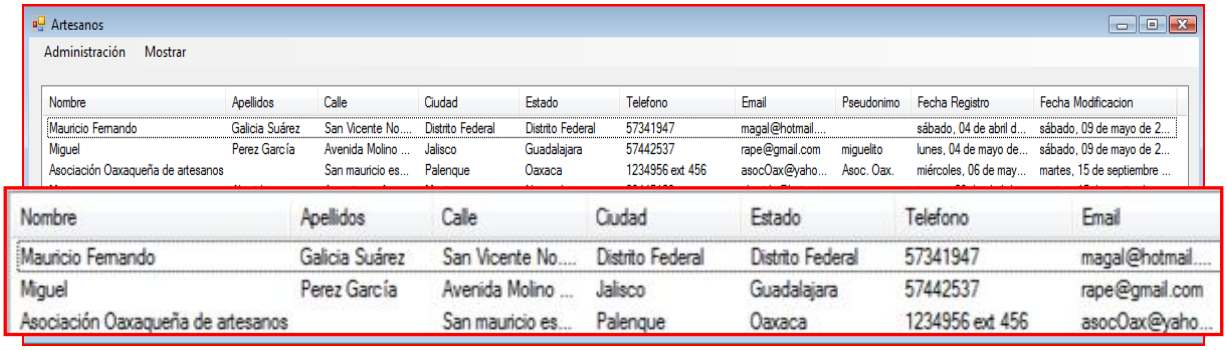
Menú para los artesanos y artesanías.



Menú para los clientes y órdenes.

5.5.4.1.1 Artesanos.

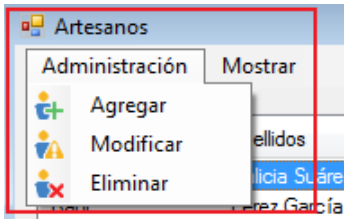
La siguiente imagen corresponde a la ventana que contiene a los artesanos de la base de datos al dar clic sobre la opción Artesanos del menú:



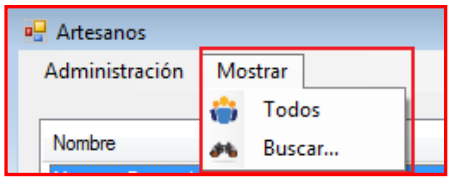
Ventana de artesanos.

5.5.4.1.1.1 Menú interfaz de artesanos.

Las siguientes imágenes corresponden al menú de la ventana de artesanos:



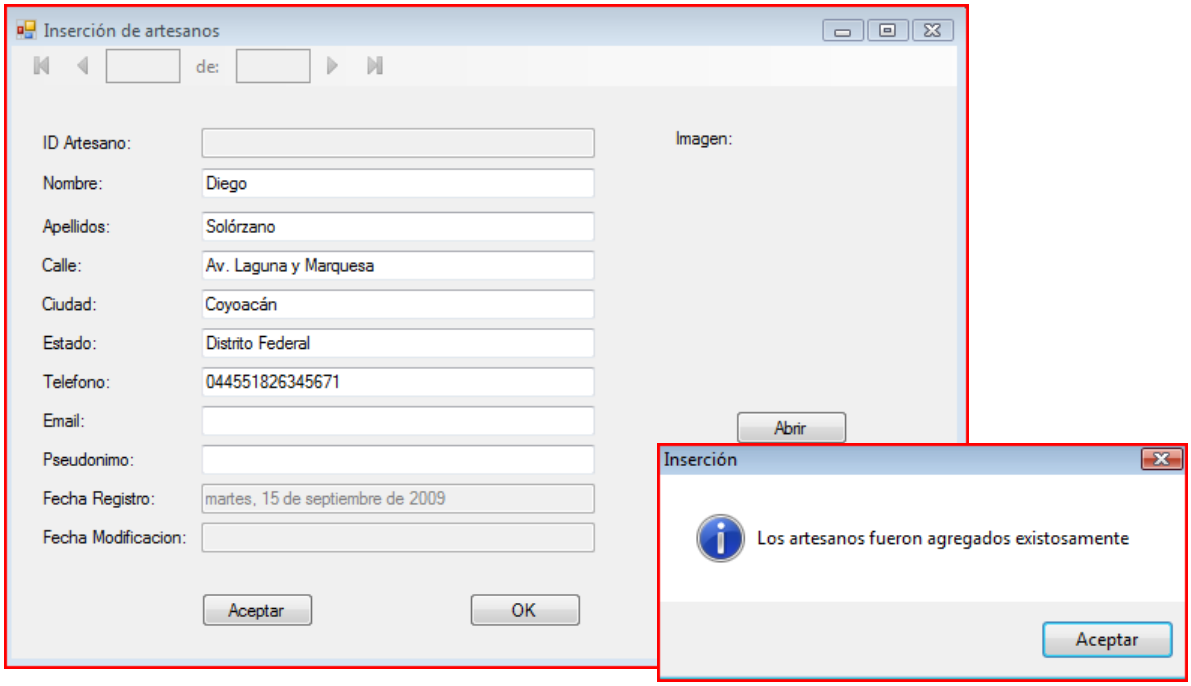
Menú para agregar, modificar y eliminar.



Menú para mostrar todos y buscar.

5.5.4.1.1.2 Inserción de artesanos.

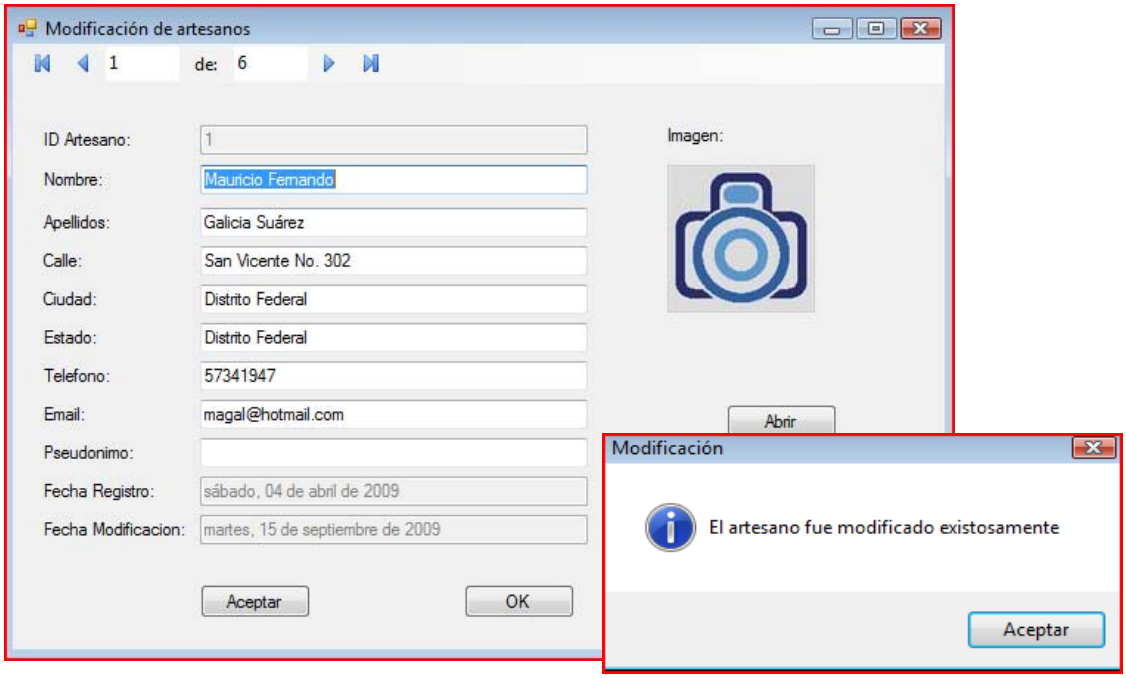
En una operación de inserción de artesanos, aparecerá una ventana como la de la siguiente imagen con los campos vacios para poder insertar los datos, incluyendo la imagen. Si la operación fue exitosa aparecerá un cuadro de mensaje como el que aparece también en la imagen sobre la ventana:



Inserción de artesanos.

5.5.4.1.1.3 Modificación Artesanos.

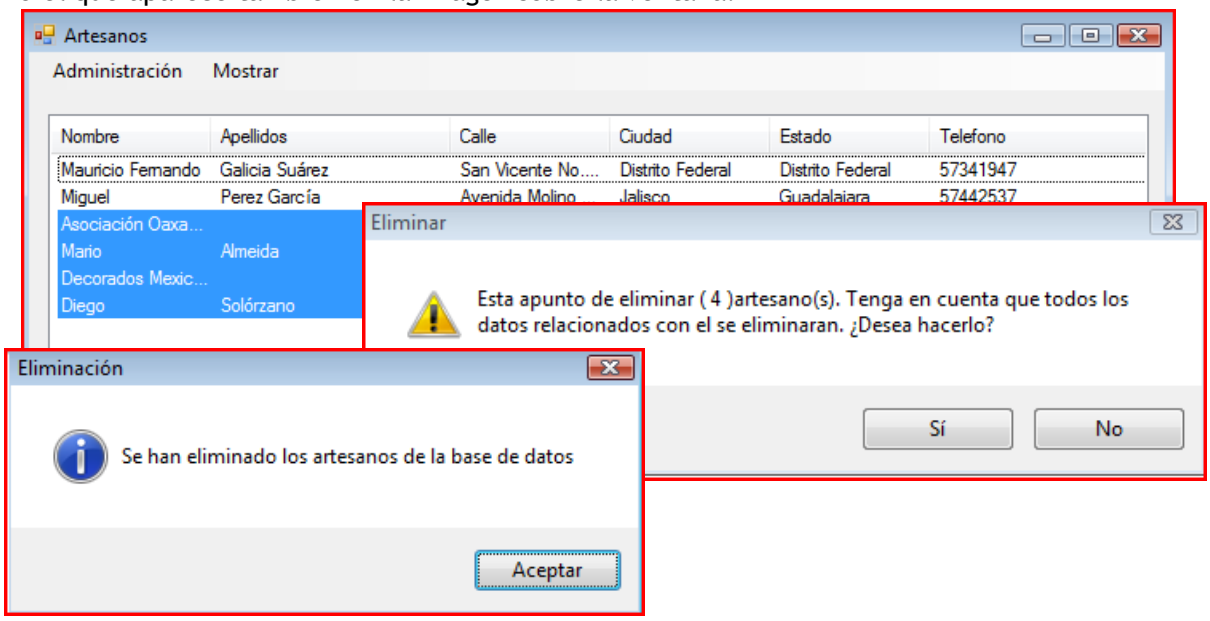
En una operación de modificación de artesanos, aparecerá una ventana como la de la siguiente imagen con los datos del artesano seleccionado de la ventana de artesanos, incluyendo la imagen. Si la operación fue exitosa aparecerá un cuadro de mensaje como el que aparece también en la imagen sobre la ventana:



Modificación de artesanos.

5.5.4.1.1.4 Eliminación de Artesanos.

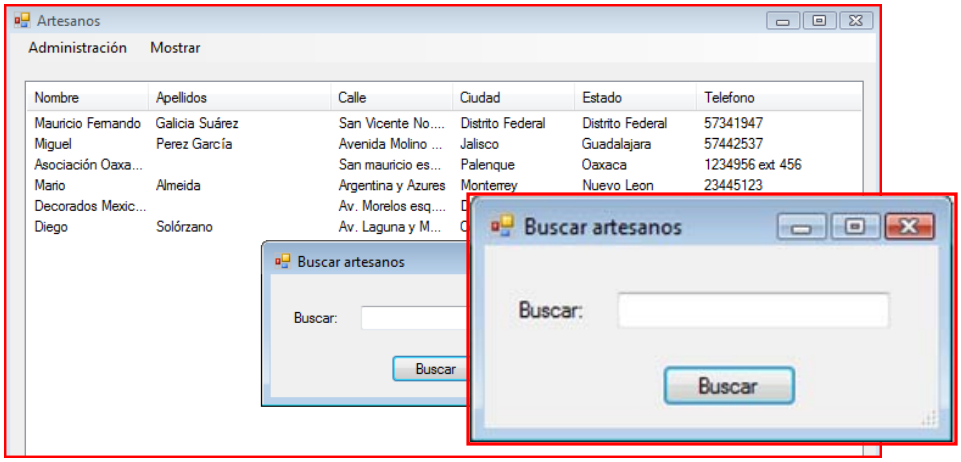
En una operación de eliminación de artesanos, se seleccionan los registros a eliminar como en la ventana de la siguiente imagen. Con lo cual aparecerá un cuadro de mensaje como el que también se muestra. Si se acepta, los datos será eliminados y aparecerá un cuadro de mensaje como el que aparece también en la imagen sobre la ventana:



Eliminación de artesanos.

5.5.4.1.1.5 Búsqueda de artesanos.

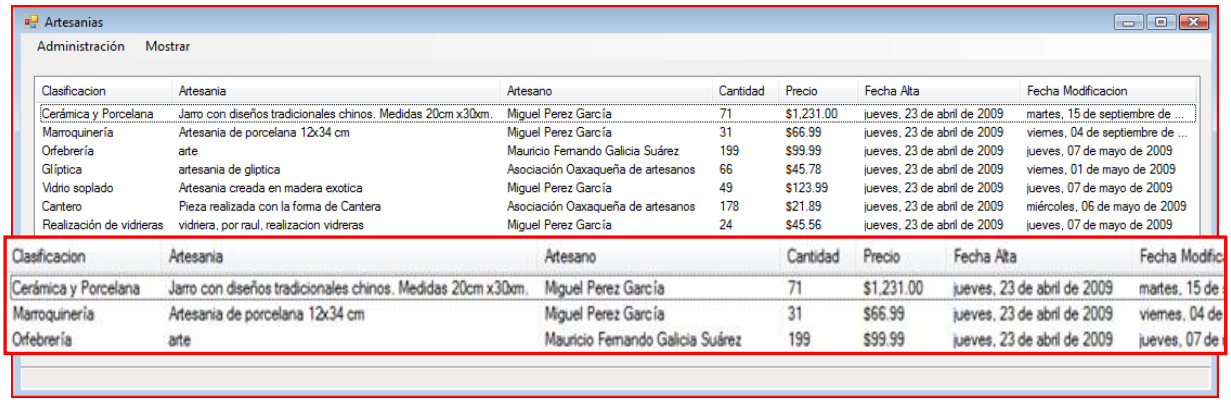
En una búsqueda de artesanos, aparecerá una nueva ventana como el que se muestra en la siguiente imagen en donde se colocará el nombre o parte de él para realizar la búsqueda de los artesanos:



Búsqueda de artesanos.

5.5.4.1.2 Artesanías.

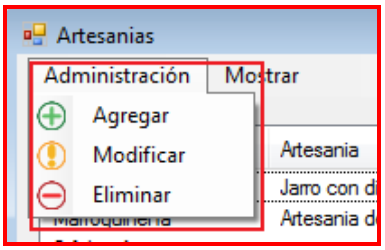
La siguiente imagen corresponde a la ventana que contiene a las artesanías de la base de datos al dar clic sobre la opción Artesanías del menú:



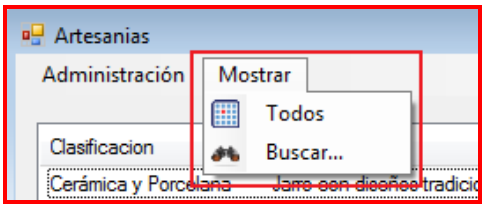
Ventana de artesanías.

5.5.4.1.2.1 Menú interfaz de artesanías.

Las siguientes imágenes corresponden al menú de la ventana de artesanías:



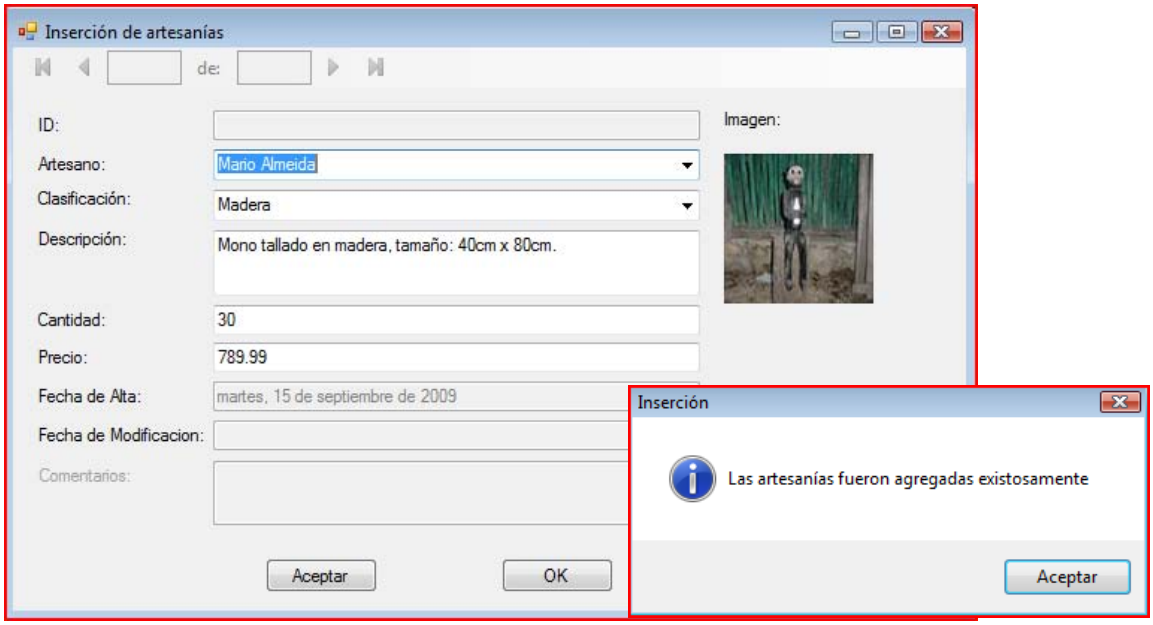
Menú para agregar, modificar y eliminar.



Menú para mostrar todos y buscar.

5.5.4.1.2.2 Inserción de artesanías.

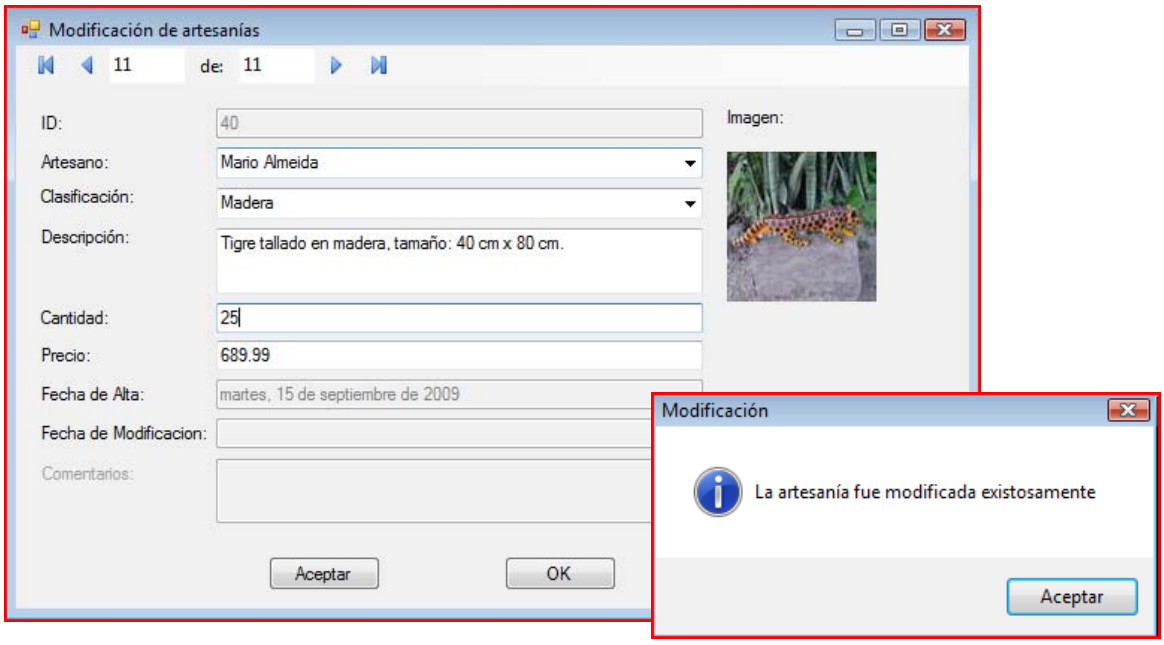
En una operación de inserción de artesanías, aparecerá una ventana como la de la siguiente imagen con los campos vacios para poder insertar los datos, incluyendo la imagen. Si la operación fue exitosa aparecerá un cuadro de mensaje como el que aparece también en la imagen sobre la ventana:



Inserción de artesanías.

5.5.4.1.2.3 Modificación artesanías.

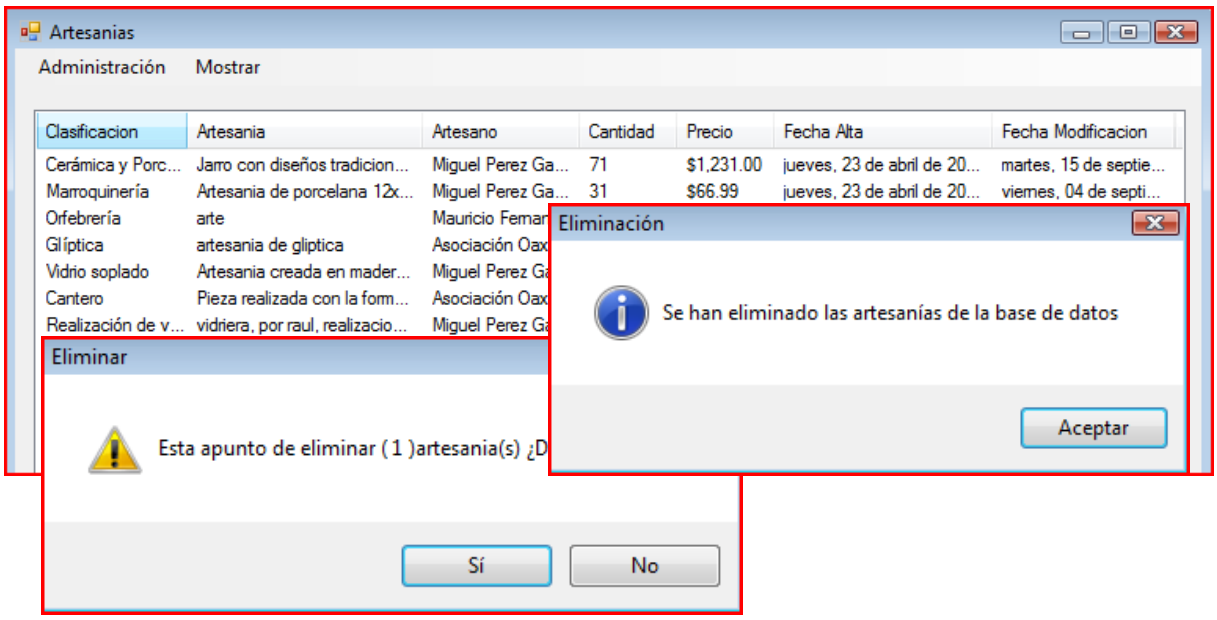
En una operación de modificación de artesanías, aparecerá una ventana como la de la siguiente imagen con los datos de la artesanía seleccionada de la ventana de artesanías, incluyendo la imagen. Si la operación fue exitosa aparecerá un cuadro de mensaje como el que aparece también en la imagen sobre la ventana:



Modificación de artesanías

5.5.4.1.2.4 Eliminación de artesanías.

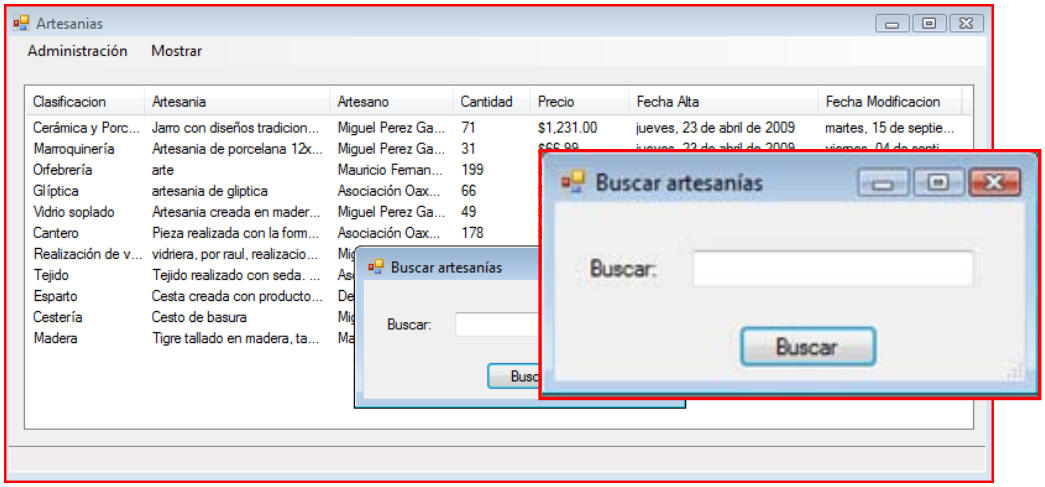
En una operación de eliminación de artesanías, se seleccionan los registros a eliminar como en la ventana de la siguiente imagen. Con lo cual aparecerá un cuadro de mensaje como el que se también se muestra. Si se acepta, los datos será eliminados y aparecerá un cuadro de mensaje como el que aparece también en la imagen sobre la ventana.



Eliminación de artesanías.

5.5.4.1.2.5 Búsqueda de artesanías.

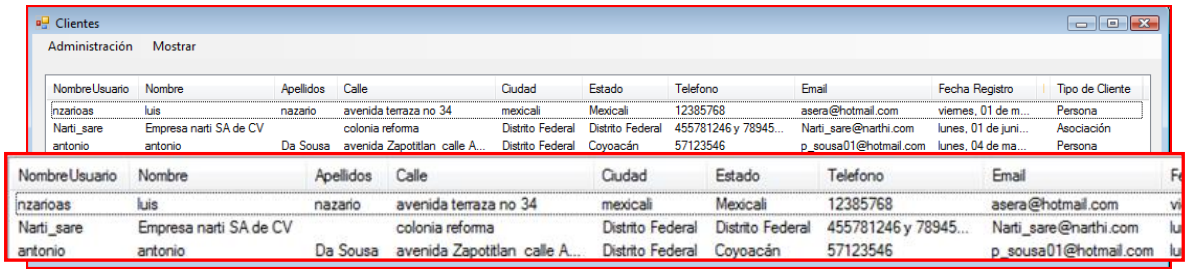
En la búsqueda de artesanías, aparecerá una ventana como el que se muestra en la siguiente imagen en donde se colocará el nombre o parte de él para realizar la búsqueda de las artesanías:



Búsqueda de artesanías.

5.5.4.1.3 Clientes.

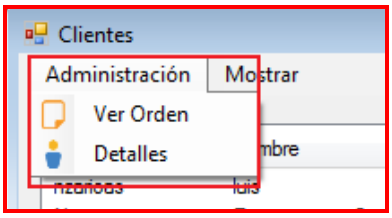
La siguiente imagen corresponde a la ventana que contiene a las artesanías de la base de datos al dar clic sobre la opción Clientes del menú:



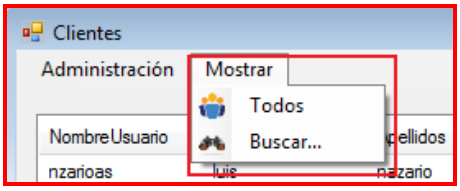
Ventana de clientes.

5.5.4.1.3.1 Menú interfaz de clientes.

Las siguientes imágenes corresponden al menú de la ventana de clientes:



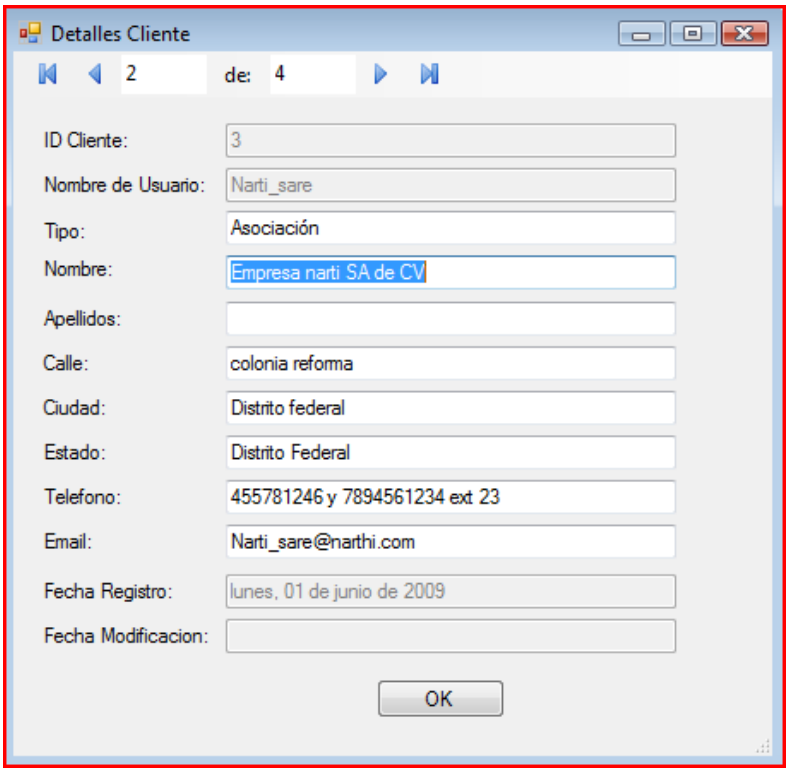
Menú para ver las ordenes del cliente.



Menú para mostrar todos y buscar.

5.5.4.1.3.2 Navegación sobre clientes.

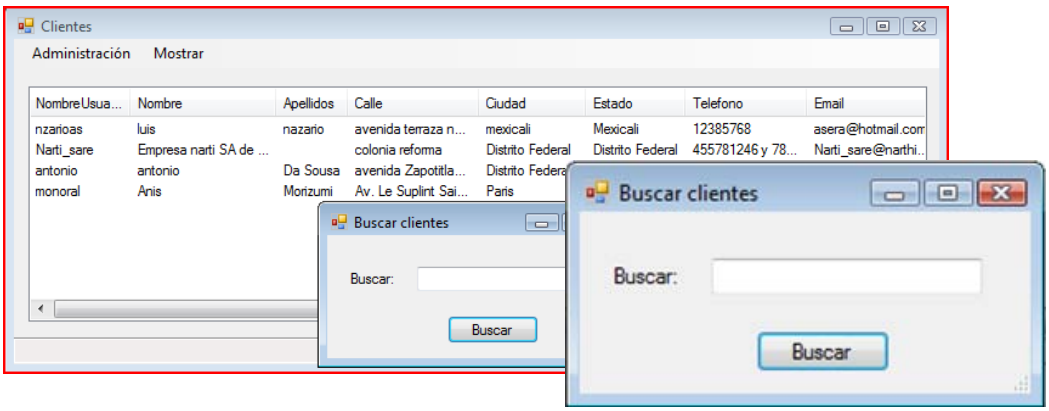
Para poder observar a detalle cada cliente localizado en la lista de la venta de clientes, se tiene la siguiente imagen que muestra la ventana que corresponde a esta función:



Navegación sobre clientes.

5.5.4.1.3.3 Búsqueda de clientes.

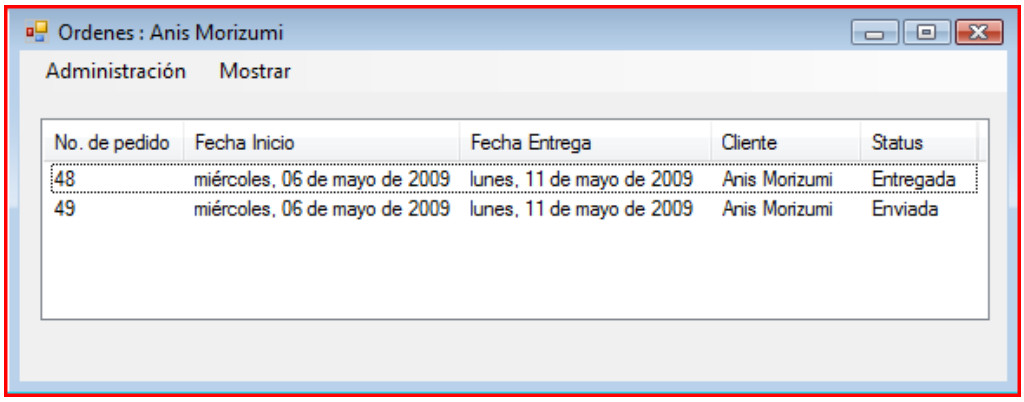
En la búsqueda de clientes, aparecerá una ventana como el que se muestra en la siguiente imagen en donde se colocará el nombre o parte de él para realizar la búsqueda de los clientes:



Búsqueda de clientes.

5.5.4.1.3.4 Ver órdenes de cliente.

La siguiente imagen presenta a la ventana encargada de mostrar las órdenes que realizó cierto cliente en la aplicación web de ArteOnLine, como son el número de la orden, la fecha en que lo realizó, fecha de entrega, y el satus en el que se encuentra cada orden:



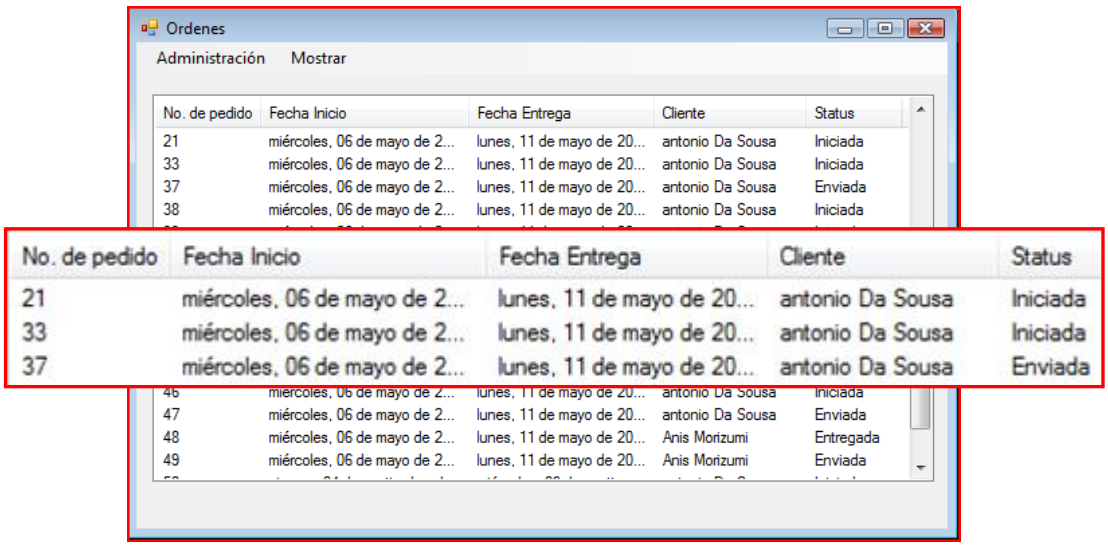
Ordenes del cliente.

5.5.4.1.4 Órdenes.

Esta sección corresponde a la ventana que contiene a todas las órdenes de la base de datos al dar clic sobre la opción Órdenes del menú.

5.5.4.1.4.1 Ver órdenes de todos los clientes.

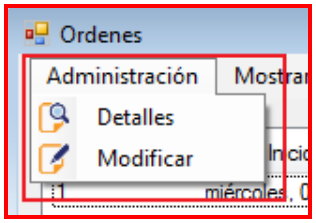
En la siguiente imagen se muestra la ventana con todas las órdenes que realizaron los clientes, la cual es la misma que la presentada anteriormente:

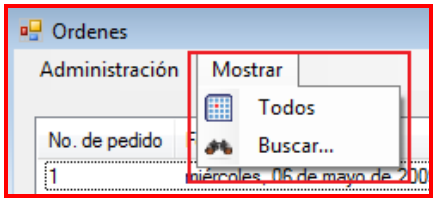


Ventana on todas las órdenes realizadas por los clientes.

5.5.4.1.4.2 Menú interfaz de clientes.

Las siguientes imágenes corresponden al menú de la ventana de órdenes:

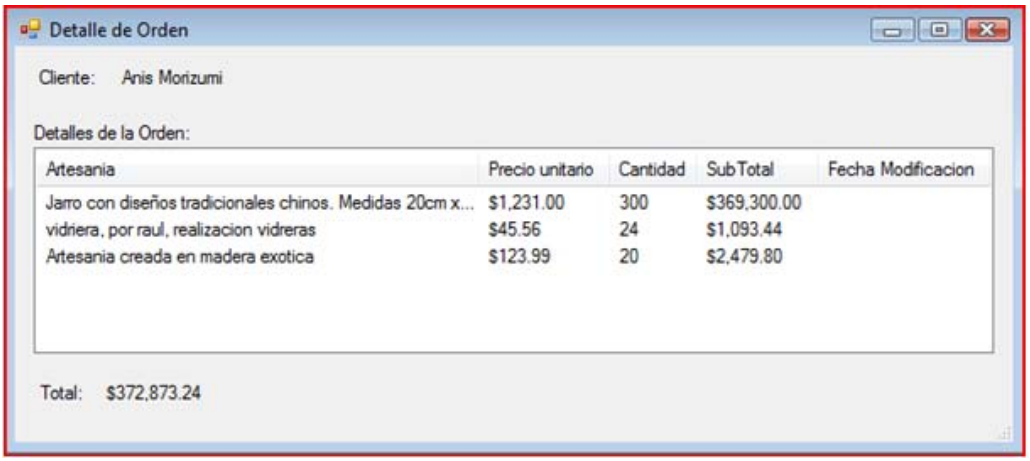




Menú para ver detalles de la orden y modificar status. ***Menú para mostrar todos.***

5.5.4.1.4.3 Ver detalles de la orden realizada.

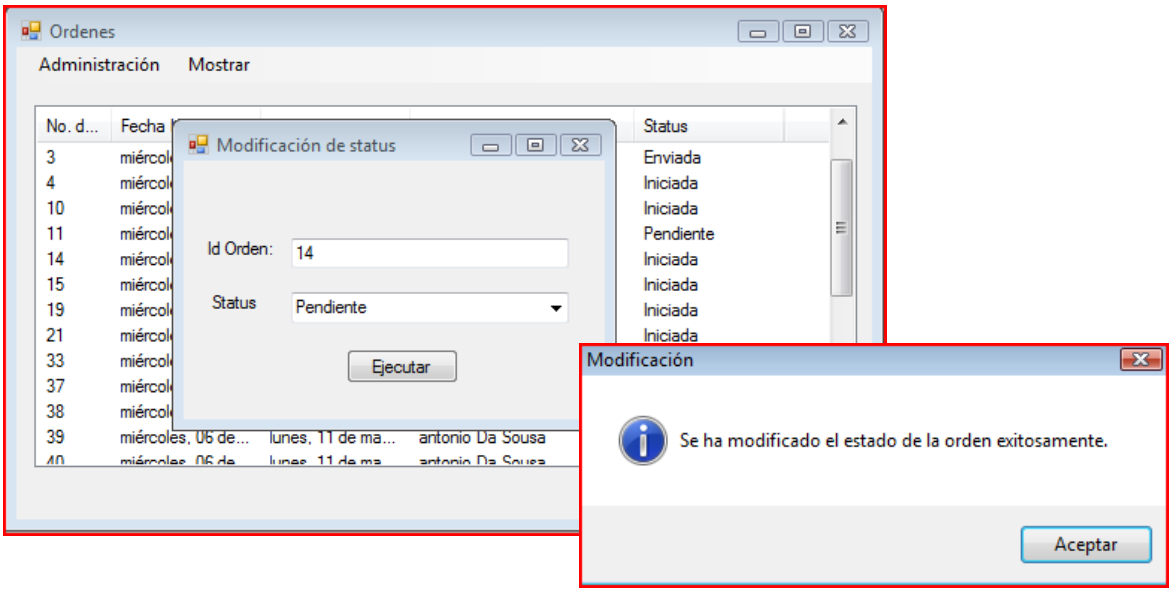
En la siguiente imagen se observa una ventana cuando se quiere modificar una orden en específico que se haya seleccionado en la lista de la ventana de las órdenes:



Ventana con los detalles de una orden.

5.5.4.1.4.4 Modificar status de orden.

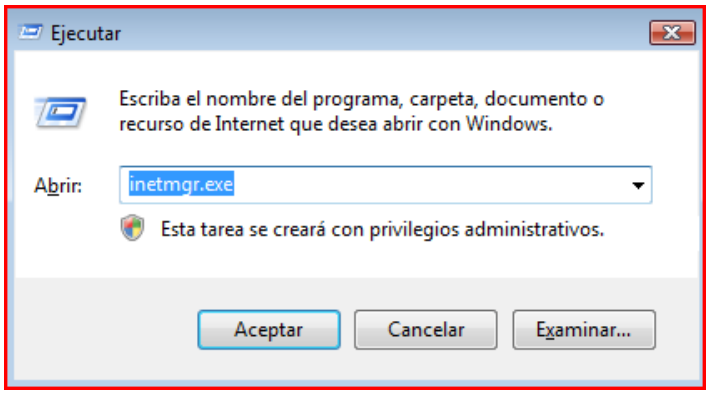
En una operación de modificación del status de una orden, aparecerá una ventana como la de la siguiente imagen con el id de la orden a modificar, junto con las opciones para el status. Si la operación fue exitosa aparecerá un cuadro de mensaje como el que aparece también en la imagen sobre la ventana:



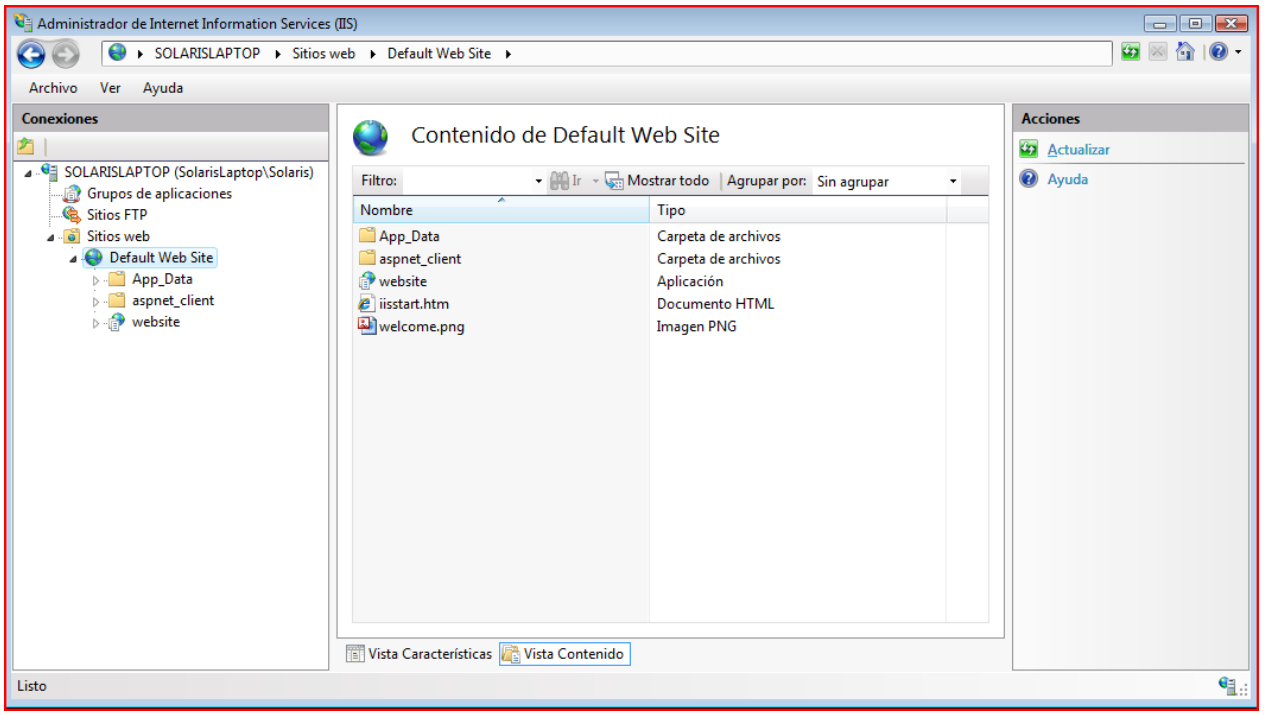
Modificación de status de una orden.

5.5.4.2 Aplicación ASP.NET.

Para poder publicar la aplicación web *ASP.NET* de *ArteOnLine* sobre un servidor y tener disponible las páginas del mismo, se hará uso de *IIS (Internet Information Sevices)7.0*, el cual viene incluido en el sistema operativo *Windows Vista Ultimate*.En primera instancia se inicia el servicio inetmgr.exe:

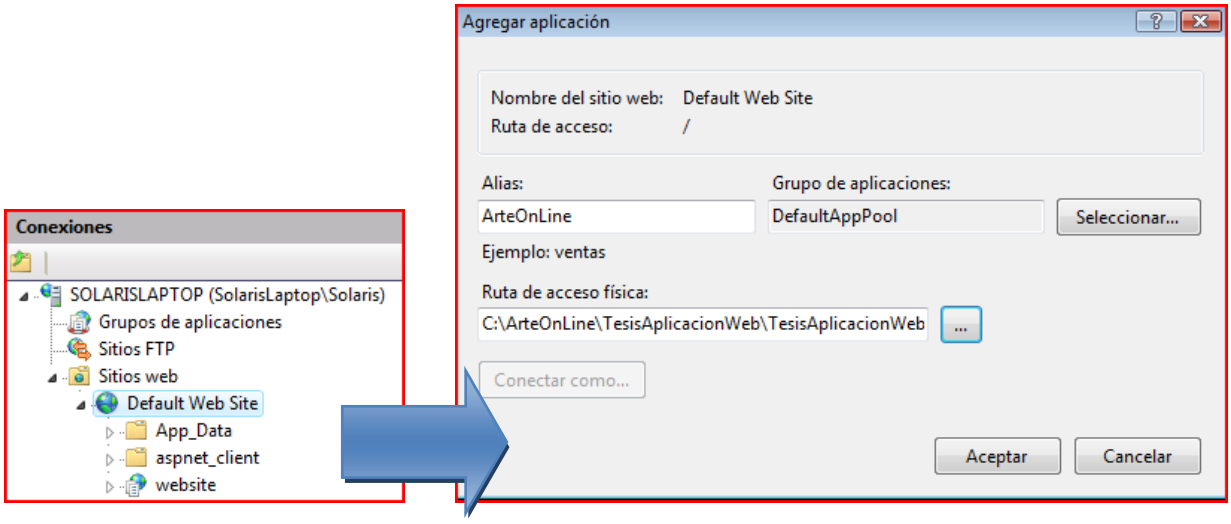


Con lo cual aparecerá el administrador de IIS como se ve en la siguiente imagen:



Administrador de IIS

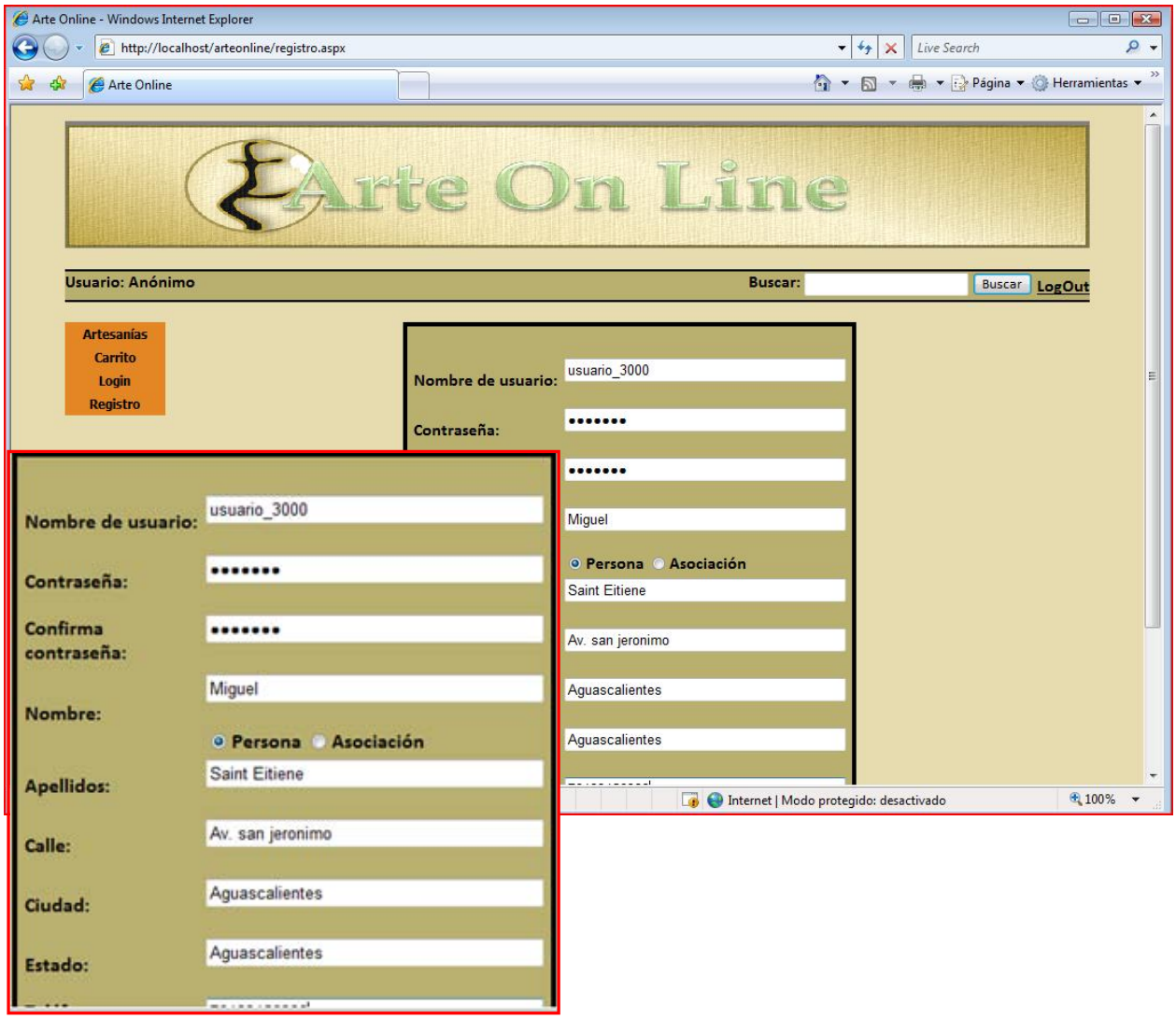
Para poder agregar la aplicación ASP.NET se debe situar sobre el lado izquierdo en Conexiones, sobre la opción Sitios Web y posteriormente en Default Web Site, una vez situados en esta opción dar clic derecho y escoger la opción Agregar Aplicación para poder agregar la aplicación ASP.NET:



Una vez realizada la publicación la página default de ArteOnLIne se expondrá en la dirección http://localhost/ArteOnLine/

5.5.4.2.1 Registro de usuario.

En la siguiente imagen se muestra la página en la que se realiza el proceso de registro de un usuario con valores válidos, esto se realiza a través de la opción Registro del menú que se localiza en parte lateral izquierda de cada página de la aplicación Web:

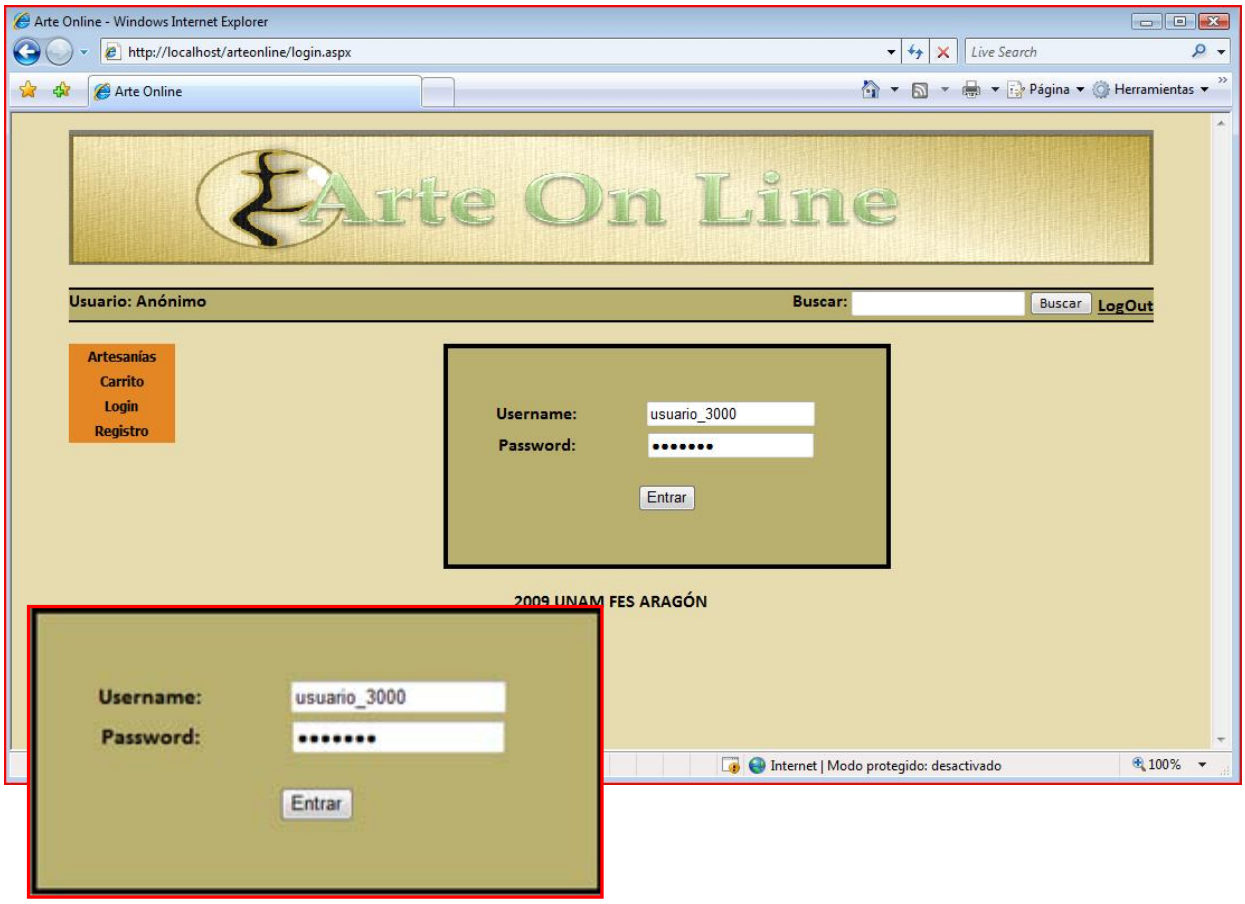


Registro de usuario.

Cuando sucede con éxito el registro del usuario, se le redirige al usuario hacia la página de login, para que coloque los datos de nombre de usuario y password que haya colocado al momento de registrase.

5.5.4.2.2 Login.

En la siguiente imagen se muestra la página en la que se realiza el proceso de Login de un usuario con valores válidos, esto se realiza a través de la opción Login del menú que se localiza en parte lateral izquierda de cada página de la aplicación Web:



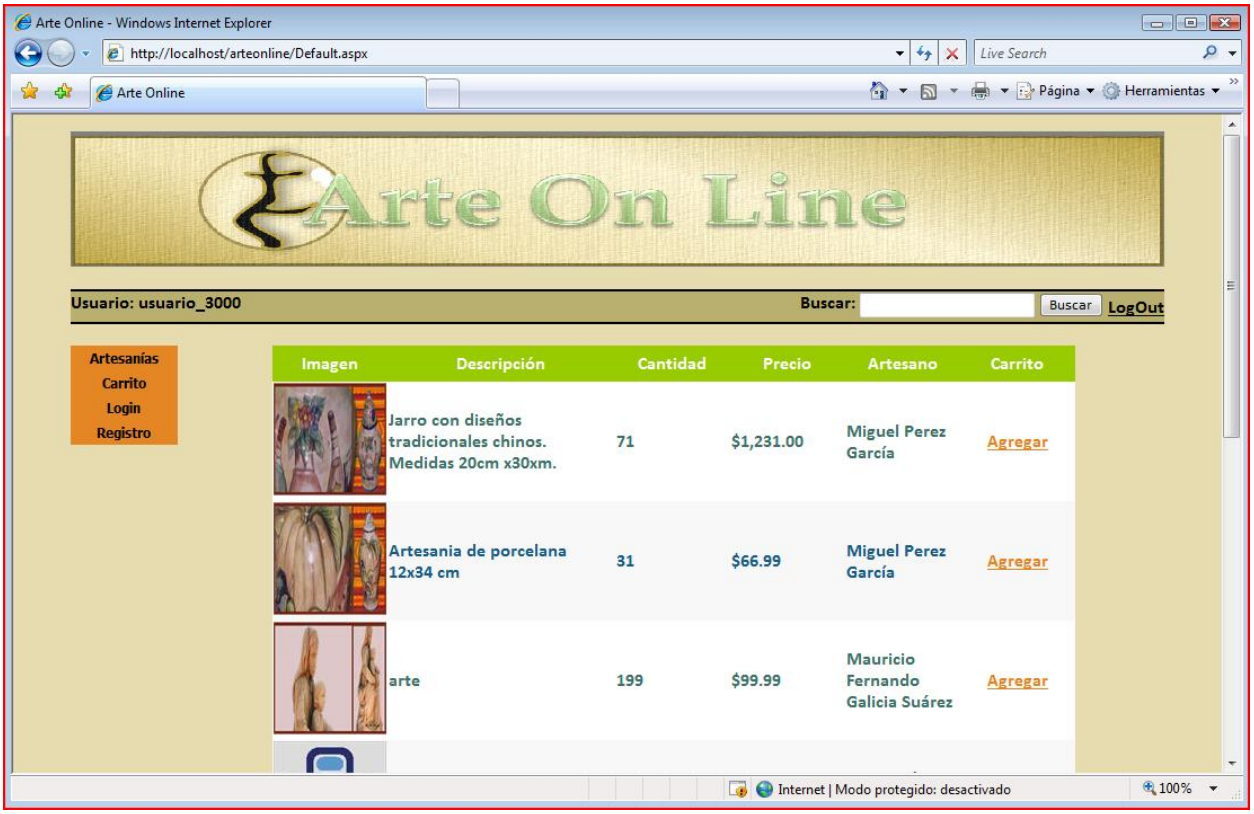
Login de usuario.

Si los datos que se dieron no corresponden con los que se dieron al momento del registro del usuario, se muestra un texto como el siguiente:

Datos de usuario incorrectos.

5.5.4.2.3 Artesanías.

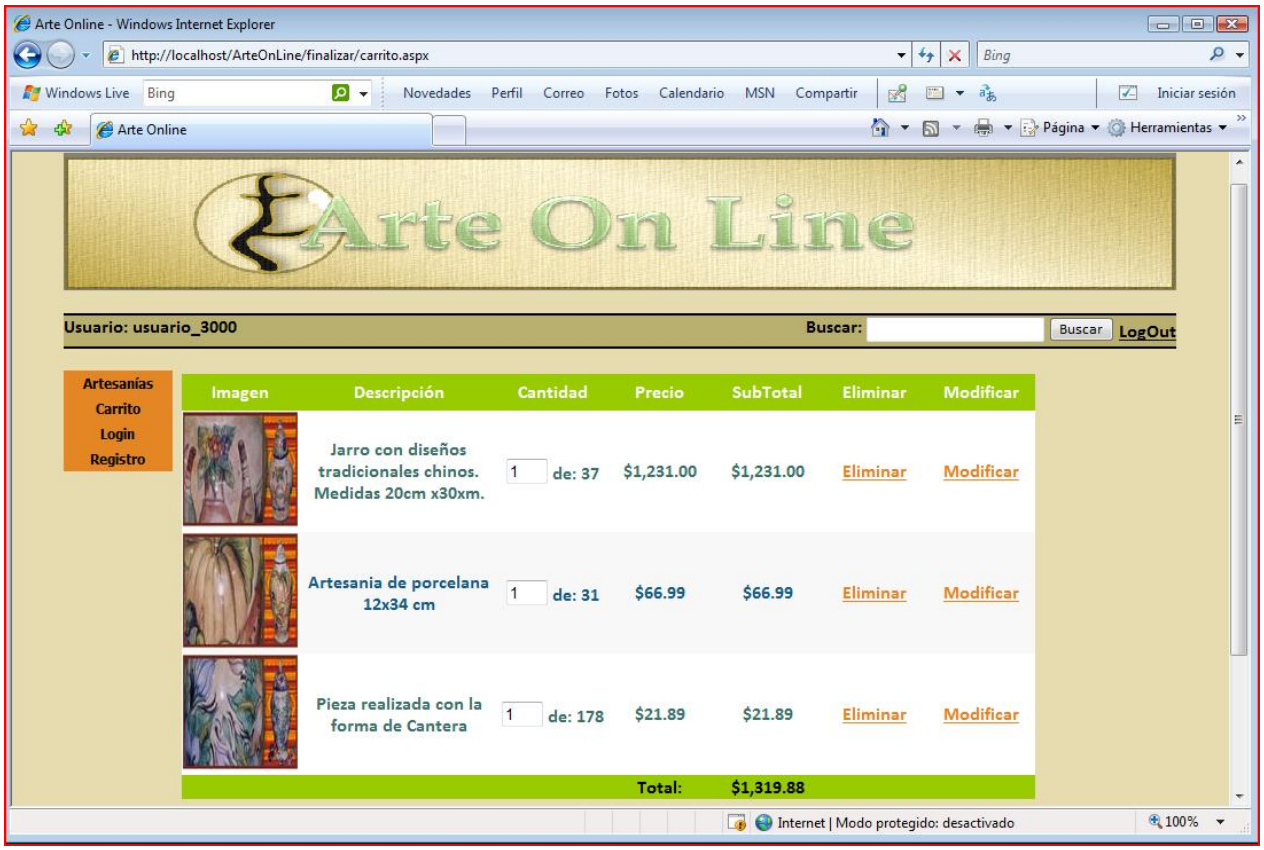
En la siguiente imagen se muestra la página en la que se muestran las artesanías a través de la opción Artesanías del menú que se localiza en parte lateral izquierda de cada página de la aplicación Web:



Artesanías.

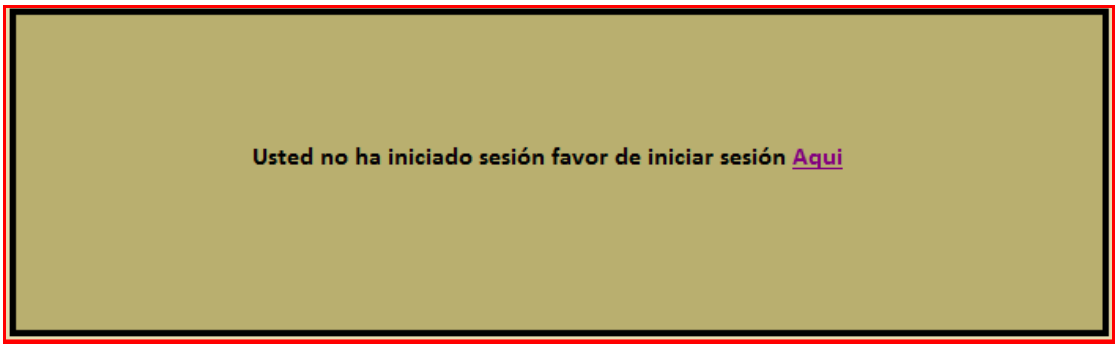
5.5.4.2.4 Agregado de elementos al carrito de compa.

Cuando se da clic en la opción Agregar localizada en cada artesanía que se muestra en la página de artesanías, se realiza el proceso de agregar el elemento al carrito de compra, si el usuario ha iniciado sesión se realiza tal proceso, por lo que se puede ir a la página del carrito a través del menú Carrito de la página, por lo que se puede observar los objetos agregados como en la siguiente imagen:



Carrito de compra.

Si no se ha iniciado sesión y se quiere agregar el elemento al carrito, se mostrará una página con el mensaje que se ve en la siguiente imagen:



Sin sesión iniciada.

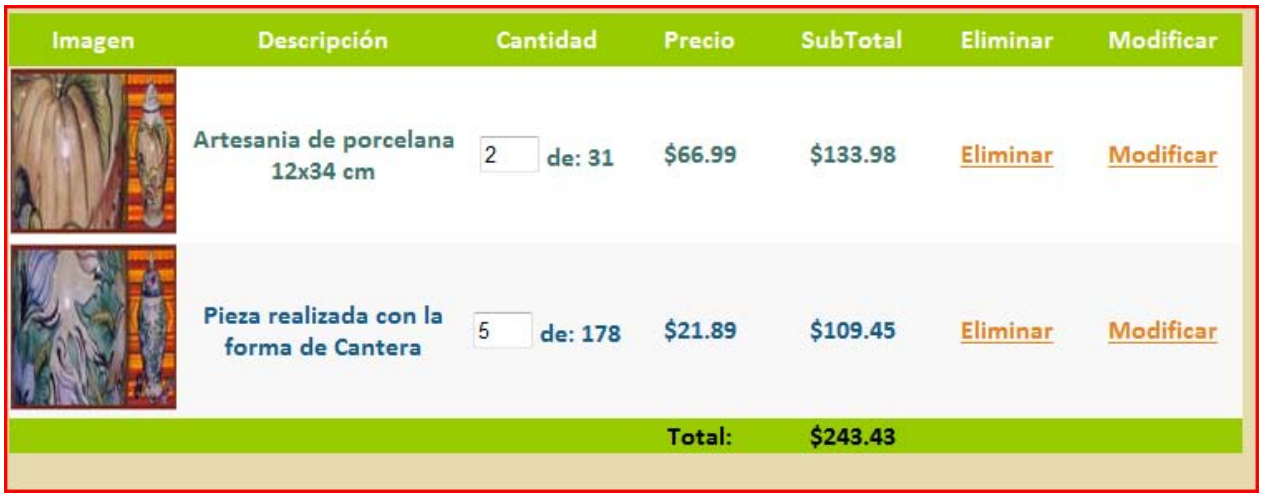
5.5.4.2.5 Modificar.

Para modificar las cantidades que se requieren de una artesanía, dar clic sobre la opción Modificar una vez que se haya colocado la cantidad deseada y en consecuencia el o las artesanías serán modificadas:

Modificación de cantidad de artesanías sobre el carrito de compra.

5.5.4.2.6 Eliminar elementos del carrito de compra.

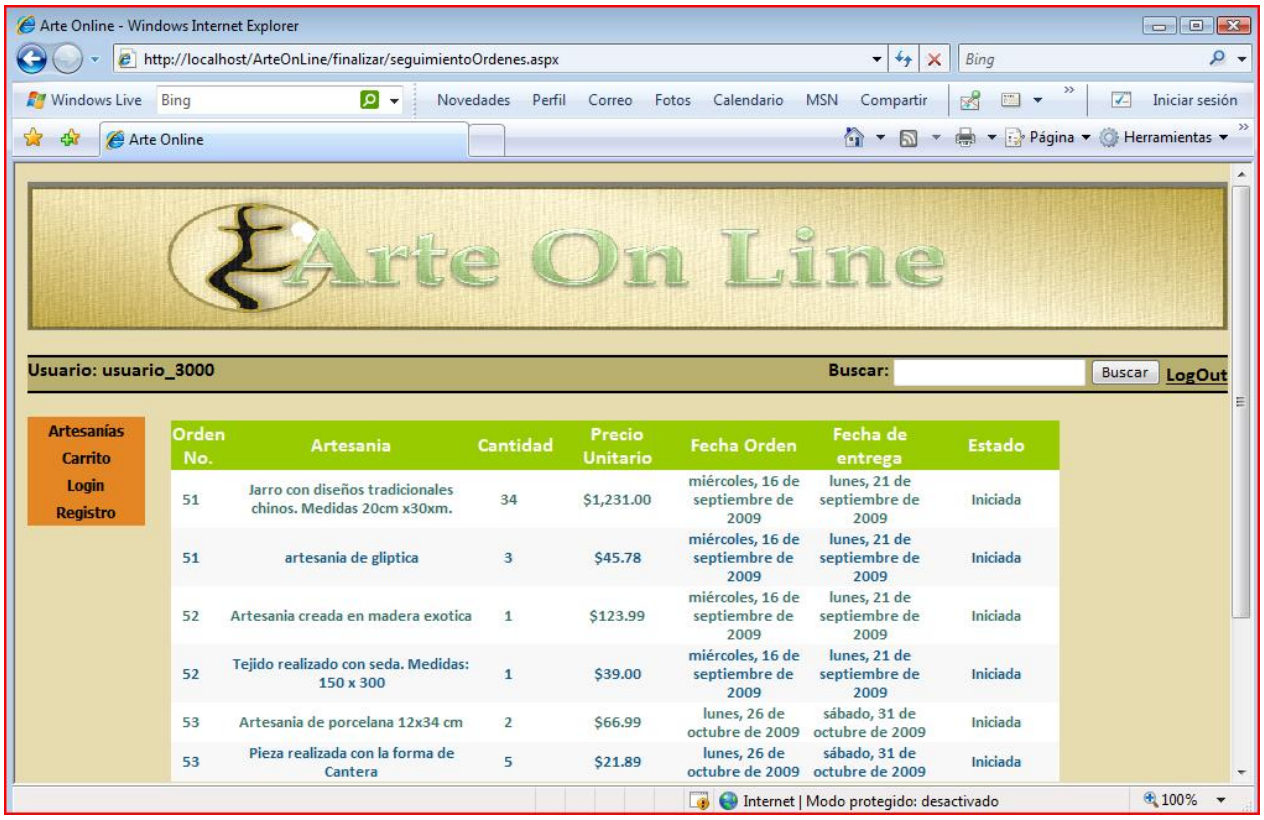
Para eliminar una artesanía, dar clic sobre la opción Eliminar sobre la artesanía deseada y en consecuencia el o las artesanías serán eliminadas:



Eliminación de elementos del carrito de compra.

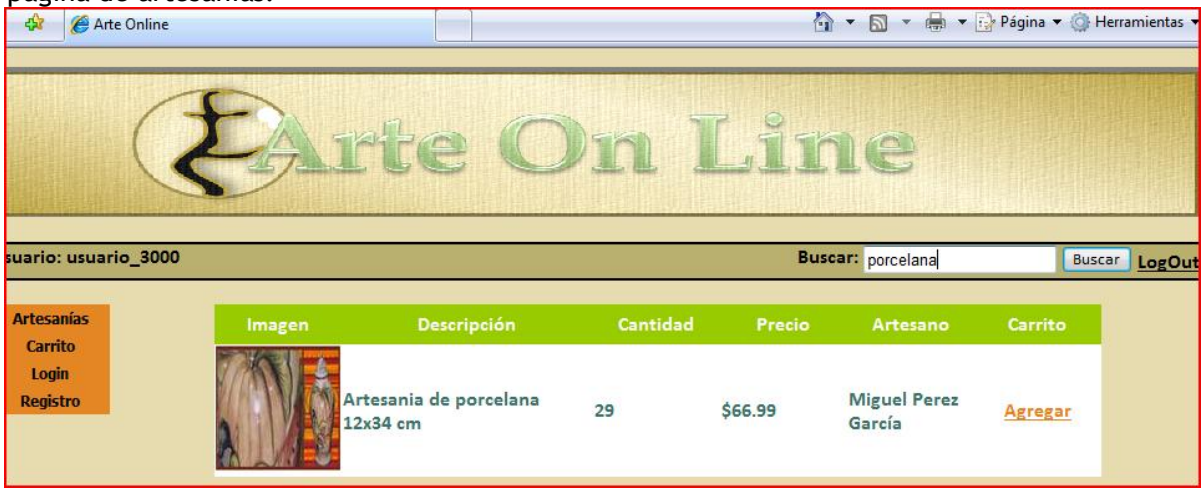
5.5.4.2.7 Creación de orden de compra y seguimiento de las órdenes.

Una vez que ya se tienen las artesanías listas, debe de darse clic sobre el botón Finalizar para crear la orden, además de que se mostrará un cuadro de texto indicando que la orden fue creada exitosamente. Posteriormente, si se requiere, se pueden ver las órdenes:



5.5.4.2.8 Búsqueda.

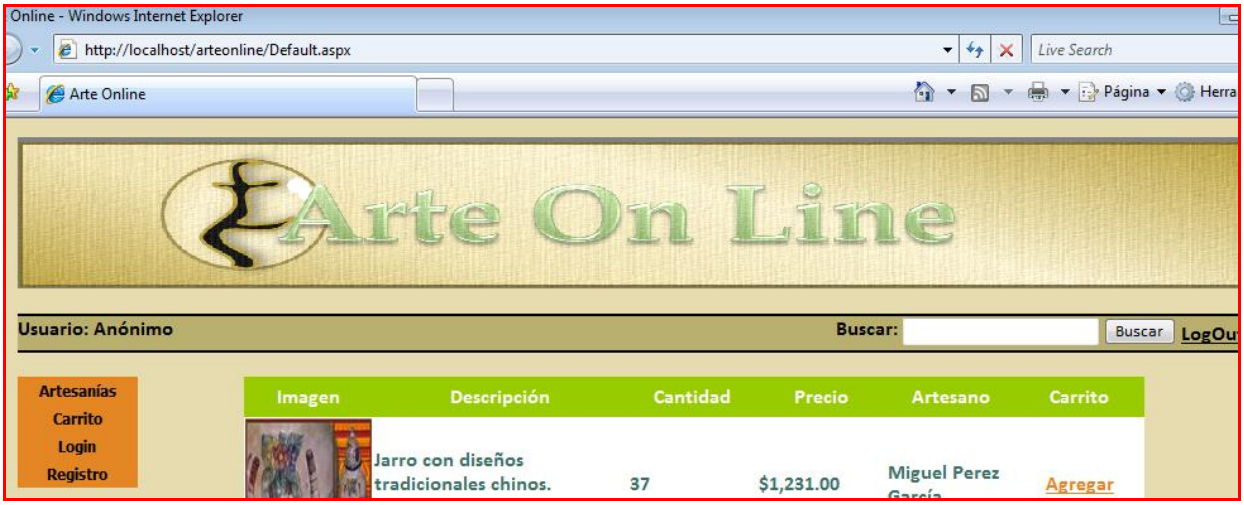
Para poder realizar la búsqueda de artesanías debe colocarse el o parte del nombre de la(s) artesanía(s) a buscar sobre el campo localizado en la parte superior de cada página, posteriormente dar clic sobre el botón de búsqueda y el resultado de la búsqueda será mostrado en la página de artesanías:



Resultado de la búsqueda.

5.5.4.2.9 Logout.

Para poder terminar la sesión del usuario, dar clic sobre la opción Logout localizado en la parte derecha de la opción de búsqueda:



Término de sesión.

CONCLUSIÓN.

En esta tesis se observó como *LINQ* consolida la forma en que se lleva a cabo la implementación de consultas sobre los datos que son manipulados a través del lenguaje de programación, disponiendo para ello de las diversas extensiones existentes para LINQ. De esta forma, *LINQ* amplia las definiciones que tienen los lenguajes de programación que estén habilitados para poder ejecutar *LINQ* de *.NET Framework*, además de que agrega más elementos a la infraestructura adyacente. También, como se vio, es posible realizar procesos de consulta que requieran de la obtención de los datos a través de diferentes formas de consulta sobre las fuentes de datos en unas cuantas líneas de código, dejando al programador implementar la lógica del negocio de una forma más transparente. Lo que se traduce en lo siguiente:

- Las operaciones que se realizan resultan más concisas y legibles, ya que se permite en una misma consulta, realizar varias operaciones de selección para poder filtrar los resultados que se requieren de acuerdo a varias condiciones, las cuáles dependen del programador.

- Se logra, en una operación de filtrado, reducir a un nivel mínimo el código implementado en la aplicación que represente a la lógica de negocio, lo que a su vez se verá reflejado en un eficiente ordenamiento y agrupación de los datos.

- Una consulta que se realice sobre una fuente de datos específica, haciendo uso de una de las extensiones de *LINQ*, en este caso *LINQ To SQL*, puede ser trasladada a otros ámbitos de trabajo sobre diversas fuentes para los datos con poca o ninguna modificación.

- El tiempo dedicado a la programación ya no ocupará un espacio prolongado en lo que respecta al manejo de los datos, ya que una consulta, como se observó, se realiza en unos cuantos pasos.

Para los procesos de consulta en los que se ven involucradas transacciones, resulta más fácil dejar a *LINQ to SQL* administrar dichas transacciones y en las cuáles se pueden escoger entre los tres modelos que se presentaron en esta tesis acerca de las transacciones, aunque en este caso es decisión del programador saber a cual referirse. De esta forma las operaciones de consulta que se realizan específicamente sobre los datos provenientes de la base de datos son más simples y que no requieren de aplicar, por ejemplo, las diversas API's que contienen clases y servicios para el acceso a datos sobre las bases de datos como lo son *ADO.NET* de *.NET Framework* o *JDBC* de *Java*, puesto que la forma en cómo implementan el manejo de las consultas sobre estos datos, resulta totalmente distinta.

Otra característica importante es la portabilidad, aunque en este rubro no existe un problema, ya que la plataforma de la que deriva, fue diseñada con ese propósito, por lo que una aplicación basada sobre la plataforma de *.NET Framework* y en la que se ejecutan consultas basadas sobre *LINQ*, operará de igual manera entre diversas arquitecturas. Actualmente existe *Mono*, el cual es una implementación de código abierto de *.NET Framework* de *Microsoft*, y que está basada en los estándares *ECMA* e ISO (*ECMA* 334/ISO 23270 para *C#* y *ECMA* 335/ ISO 23271 para los estándares que definen al *CLI* y el *CLR)* y con lo que se ofrece soporte para esta plataforma. Por lo que para la versión de *Mono* 2.6 (Noviembre de 2009) se tendrá soporte para una integración de *LINQ* con las bases de datos ya que en la versión 2.4 se tiene soporte para las definiciones en *C#* 3.0, esto incluye a los elementos que forman parte de *LINQ*.

Sin embargo para poder usar *LINQ*, como en cualquier otro caso, es necesario comprender las bases o conceptos sobre las que se fundamenta, ya que esto permitirá, en algunos casos, comprender los errores de programación que puedan presentarse y en otros, aprovechar las

características para mejorar la producción y rendimiento en las aplicaciones de software. El aspecto importante a tomar en consideración en esta tesis, es el nivel de profundidad que se le dé, en lo que respecta a *LINQ*, para poderlo aprovechar al máximo, ya que en esta tesis solo se presentan los elementos más importantes de esta tecnología para poder hacer un uso inmediato y rápido sobre aplicaciones como se vio en la parte final de esta tesis. Con esto se puede decir que objetivo final de mostrar qué es LINQ y como soluciona los problemas existentes entre el lenguaje de programación y los datos resultó satisfactoria, así como también el hacer el uso de LINQ to SQL aunado a un buen diseño para alcanzar un mejor nivel en el proceso de desarrollo de software, y con esto solventar los problemas de complejidad en cuanto al manejo de los datos relacionales.

BIBLIOGRAFÍA.

Referencias.

1. MSDN: Microsoft Development:
http://msdn.microsoft.com

2. Diseñador de objetos relacionales:
http://msdn.microsoft.com/esmx/library/bb384429.aspx

3. The .NET Standard Query Operators:
http://msdn.microsoft.com/en-us/library/bb394939.aspx

4. C# Programming Guide:
http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx

5. LINQ to SQL: .NET Language-Integrated Query for Relational Data:
http://msdn.microsoft.com/en-us/library/bb425822.aspx

6. Microsoft .NET:
http://es.wikipedia.org/wiki/Microsoft_.NET

7. Mono:
http://www.mono-project.com

8. Quaere:
http://quaere.codehaus.org

9. JoSQL:
http://josql.sourceforge.net

10. Devart:
http://www.devart.com/

11. DbLinq Project:
http://code2code.net/DB_Linq/

Libros.

1. Pro C# with .NET 3.0.
Andrew Troelsen
Ed. Apress, 2007.

2. Cross-plataform .NET Development. Using Mono, portable .NET and Microsoft .NET.
M. J. Easton and Jason King.
Edit. Apress, 2004.

3. Accelerated C# 2008.
   Trey Nash.
   Edit. Apress, 2007.

4. Pro LINQ object relational mapping with C# 2008.
   Vijay P. Meth.
   Edit. Apress, 2008.

5. Pro Java.
   Brett Spell.
   Edit. Apress, 2007.

6. Beginning C# 2008 Data Bases.
   Vidya Vraf Agarival and James Huddleston.
   Edit. Apress, 2008.

7. Pro C# and the .NET 3.5 Plataform.
   Andrew Troelsen.
   Edit. Apress, 2007.

ANEXO A.

A.1 DETALLE DE CASOS DE USO APLICACIÓN WINDOWS FORMS.

A.1.1 Detalle del Caso de Uso "Agregar Artesano"

Definición	
Caso de uso:	**Agregar Artesano**
Tipo:	Básico.
Actor principal:	***Usuario***
Descripción:	Este caso de uso describe como el *usuario* puede agregar un artesano en la BD.
Pre-condiciones:	El Usuario debe iniciar el sistema.
Pos-condiciones:	El Usuario podrá agregar a un artesano.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos				
Flujo básico:				
Agregar Artesano				
Inicio de Ventana normal				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanos en el menú Local.	2.	El sistema muestra una ventana donde se muestran los Artesanos que se encuentran en la Base de datos	E1
3	El usuario opta por agregar un artesano a la base de datos y da clic en el menú Administrar→ Agregar	4	Se muestra una ventana con los campos vacíos para los datos de los artesanos.	E3
5	Si se han insertado los datos correctos y el usuario decide dar clic en el botón Aceptar, se agrega el nuevo artesano temporalmente. Los campos se limpian automáticamente.		Si el usuario ha decidido que el número de artesanos agregados es suficiente, da clic sobre el botón guardar.	

Flujos alternativos:				
Inicio de Ventana sin datos				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanos en el menú Local.	2.	El sistema muestra una ventana donde se muestran los Artesanos que se encuentran en la Base de datos	E1
3.	El usuario opta por agregar un artesano a la base de datos y da clic en el menú Administrar→ Agregar	4.	Se muestra una ventana con los campos vacíos para los datos de los artesanos.	E2
5.	Si se no se han insertado datos y el usuario decide dar clic en el botón Aceptar, aparece un mensaje de error indicando que no se pueden dejar campos vacíos.			

Inicio de Ventana con datos incorrectos

Actor		Sistema		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanos en el menú Local.	2.	El sistema muestra una ventana donde se muestran los Artesanos que se encuentran en la Base de datos.	E1
3.	El usuario opta por agregar un artesano a la base de datos y da clic en el menú Administrar→ Agregar.	4.	Se muestra una ventana con los campos vacíos para los datos de los artesanos.	E2
5.	Si los datos que se han insertado datos son incorrectos y el usuario decide dar clic en el botón Aceptar, aparece un mensaje de error indicando que el tipos de los datos no son los esperados.			

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Conexión con BD perdida	Cuadro de dialogo que muestre la siguiente leyenda: "Ha ocurrido un error al cargar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador".
E2	Formato incorrecto de datos	Cuadro de dialogo que muestre la siguiente leyenda:"Ha ocurrido un error, verifique que el tipo de los datos sean los correctos o que no hayan campos vacíos".
E3	Error al insertar los datos en la BD	Cuadro de dialogo que muestre la siguiente leyenda "Ha ocurrido un error al insertar los datos. Verifique que los datos sean los correctos".

A.1.2 Detalle del Caso de Uso "Eliminar Artesano"

Definición	
Caso de uso:	**Eliminar Artesano**
Tipo:	Básico.
Actor principal:	***Usuario***
Descripción:	Este caso de uso describe como el *usuario* puede eliminar un artesano existente en la BD.
Pre condiciones:	El Usuario debe iniciar el sistema.
Pos condiciones:	El Usuario podrá eliminar a un artesano.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos				
Flujo básico:				
Eliminar Artesano				
Eliminación de artesano normal				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanos en el menú Local.	2.	El sistema muestra una ventana donde se muestran los Artesanos que se encuentran en la Base de datos	E1
3.	El usuario opta por eliminar uno o varios artesanos de la base de datos y da clic en el menú Administrar→ Eliminar	4.	Si se han seleccionado registros de la lista mostrada, aparecerá un cuadro de mensaje advirtiendo que se procederá a borrar los registros de la base de datos.	E2

5.	Si el usuario decide de todos modos eliminarlos da clic en el botón aceptar.	6.	Si no ocurre un error, se muestra un cuadro de mensaje indicando que la operación fue exitosa.	
Flujos alternativos:				
Eliminación de artesano sin selección				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanos en el menú Local.	2.	El sistema muestra una ventana donde se muestran los Artesanos que se encuentran en la Base de datos.	E1
3	El usuario opta por eliminar uno o varios artesanos de la base de datos y da clic en el menú Administrar→ Eliminar.	4	Si no se han seleccionado registros de la lista mostrada, aparecerá un cuadro de mensaje de error indicando que no se han seleccionado los registros a eliminar.	

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Conexión con BD perdida	Cuadro de dialogo que muestre la siguiente leyenda: "Ha ocurrido un error al cargar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador".
E2	Error al eliminar los datos de la BD	Cuadro de dialogo que muestre la siguiente leyenda: "Ha ocurrido un error al eliminar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador".

A.1.3 Detalle del Caso de Uso "Modificar Artesano"

Definición	
Caso de uso:	**Modificar Artesano**
Tipo:	Básico.
Actor principal:	***Usuario***
Descripción:	Este caso de uso describe como el *usuario* puede modificar a un artesano existente en la BD.
Pre condiciones:	El Usuario debe iniciar el sistema.
Pos condiciones:	El Usuario podrá modificar a un artesano.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos				
Flujo básico:				
Modificar Artesano				
Modificación de artesano normal				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanos en el menú Local.	2.	El sistema muestra una ventana donde se muestran los Artesanos que se encuentran en la Base de datos	E1
3.	El usuario opta por modificar uno o varios artesanos de la base de datos y da clic en el menú Administrar→ Modificar.	4.	Si existen datos cargados previamente en la lista mostrada, aparecerá una ventana mostrando la información del artesano.	E3
5.	Si el usuario ha decidido que la información es la correcta, actualiza	6.	Si el usuario ha decidido terminar con la modificación de los registros da clic en el	

	el registro dando clic en el botón aceptar para seguir navegando por los registros y modificar las veces que se requiera.		botón Aceptar para cerrar la ventana y mostrar los datos en la lista actualizada.	

Flujos alternativos:				
Modificación de artesano con datos incorrectos				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanos en el menú Local.	2.	El sistema muestra una ventana donde se muestran los Artesanos que se encuentran en la Base de datos.	E1
3.	El usuario opta por modificar uno o varios artesanos de la base de datos y da clic en el menú Administrar→ Modificar o da enter con la tecla en un registro específico.	4.	Si existen datos cargados previamente en la lista mostrada, aparecerá una ventana mostrando la información del artesano.	E2
5.	Si el tipo de dato que el usuario ha insertado s incorrecto, y actualiza el registro dando clic en el botón aceptar, aparecerá un cuadro de mensaje indicando que los tipos de datos son incorrectos.			

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Conexión con BD perdida	Cuadro de dialogo que muestre la siguiente leyenda: “Ha ocurrido un error al cargar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador”
E2	Formato incorrecto de datos	Cuadro de dialogo que muestre la siguiente leyenda:"Ha ocurrido un error, verifique que el tipo de los datos sean los correctos o que no hayan campos vacíos”.
E3	Error al modificar los datos de la BD	Cuadro de dialogo que muestre la siguiente leyenda: "Ha ocurrido un error al modificar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador".

A.1.4. Detalle del Caso de Uso “Búsqueda de Artesanos”

Definición	
Caso de uso:	**Búsqueda de Artesanos**
Tipo:	Básico.
Actor principal:	***Usuario***
Descripción:	Este caso de uso describe como el *usuario* puede buscar a uno o varios artesanos existentes en la BD.
Pre condiciones:	El Usuario debe iniciar el sistema y encontrarse en la ventana de artesanos.
Pos condiciones:	El Usuario podrá buscar a una o varias artesanos.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos

Flujo básico:				
Búsqueda de Artesanos				
Búsqueda de artesano con datos				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanos en el menú Local.	2.	El sistema muestra una ventana donde se muestran los Artesanos que se encuentran en la Base de datos.	E1
3.	El usuario opta por buscar uno o varios artesanos de la base de datos y da clic en el menú Mostrar→ Buscar.	4.	Se abre una ventana con un campo vacío para colocar el nombre completo o parte de el, para buscar el artesano.	
5.	Si el usuario ha decidido que ha colocado la información correcta, y da clic en el botón buscar para iniciar el proceso de búsqueda de los artesanos por su nombre, con lo cuál se cerrará la ventana automáticamente y se podrán observar los datos en la ventana que la abrió.			

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Conexión con BD perdida	Cuadro de dialogo que muestre la siguiente leyenda: "Ha ocurrido un error al cargar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador"

A.1.5 Detalle de caso de uso "Agregar Artesanía"

Definición	
Caso de uso:	**Agregar Artesanía**
Tipo:	Básico.
Actor principal:	***Usuario***
Descripción:	Este caso de uso describe como el *usuario* puede agregar un artesanía en la BD.
Pre condiciones:	El Usuario debe iniciar el sistema.
Pos condiciones:	El Usuario podrá agregar artesanías a la base de datos.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos				
Flujo básico:				
Agregar Artesanía				
Inicio de Ventana normal				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanías en el menú Local.	2.	El sistema muestra una ventana donde se muestran las artesanías que se encuentran en la Base de datos.	E1
3	El usuario opta por agregar una artesanía a la base de datos y da clic en el menú Administrar→ Agregar	4	Se muestra una ventana con los campos vacíos para los datos de las artesanías.	E3
5	Si se han insertado los datos correctos y el usuario decide dar clic en el botón Aceptar, se agrega la nueva artesanía		Si el usuario ha decidido que el número de artesanías agregadas es suficiente, da clic sobre el botón guardar.	

	temporalmente. Los campos se limpian automáticamente.			

Flujos alternativos:				
Inicio de Ventana sin datos				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanías en el menú Local.	2.	El sistema muestra una ventana donde se muestran las artesanías que se encuentran en la Base de datos	E1
3.	El usuario opta por agregar una artesanía a la base de datos y da clic en el menú Administrar→ Agregar	4.	Se muestra una ventana con los campos vacíos para los datos de las artesanías.	E2
5.	Si se no se han insertado datos y el usuario decide dar clic en el botón Aceptar, aparece un mensaje de error indicando que no se pueden dejar campos vacíos.			

Inicio de Ventana con datos incorrectos				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanos en el menú Local.	2.	El sistema muestra una ventana donde se muestran las artesanías que se encuentran en la Base de datos	E1
3.	El usuario opta por agregar una artesanía a la base de datos y da clic en el menú Administrar→ Agregar	4.	Se muestra una ventana con los campos vacíos para los datos de las artesanías.	E2
5.	Si los datos que se han insertado datos son incorrectos y el usuario decide dar clic en el botón Aceptar, aparece un mensaje de error indicando que el tipos de los datos no son los esperados.			

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Conexión con BD perdida	Cuadro de dialogo que muestre la siguiente leyenda: ”Conexión con la BD perdida”.
E2	Formato incorrecto de datos	Cuadro de dialogo que muestre la siguiente leyenda:"Ha ocurrido un error, verifique que el tipo de los datos sean los correctos o que no hayan campos vacíos”.
E3	Error al insertar los datos en la BD	Cuadro de dialogo que muestre la siguiente leyenda “Ha ocurrido un error al insertar los datos. Verifique que los datos sean los correctos”.

A.1.6 Detalle del Caso de Uso “Eliminar Artesanía”

Definición	
Caso de uso:	**Eliminar Artesanía**
Tipo:	Básico.

Actor principal:	***Usuario***
Descripción:	Este caso de uso describe como el *usuario* puede eliminar una artesanía existente en la BD.
Pre condiciones:	El Usuario debe iniciar el sistema.
Pos condiciones:	El Usuario podrá eliminar a una artesanía.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos				
Flujo básico:				
Eliminar Artesanía				
Eliminación de artesanía normal				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanías en el menú Local.	2.	El sistema muestra una ventana donde se muestran las artesanías que se encuentran en la Base de datos	E1
3.	El usuario opta por eliminar una o varias artesanías de la base de datos y da clic en el menú Administrar→ Eliminar.	4.	Si se han seleccionado registros de la lista mostrada, aparecerá un cuadro de mensaje advirtiendo que se procederá a borrar los registros de la base de datos.	E2
5.	Si el usuario decide de todos modos eliminarlos da clic en el botón aceptar.	6.	Si no ocurre un error, se muestra un cuadro de mensaje indicando que la operación fue exitosa.	

Flujos alternativos:				
Eliminación de artesanías sin selección				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanías en el menú Local.	2.	El sistema muestra una ventana donde se muestran los Artesanos que se encuentran en la Base de datos	E1
3.	El usuario opta por eliminar una o varias artesanías de la base de datos y da clic en el menú Administrar→ Eliminar.	4.	Si no se han seleccionado registros de la lista mostrada, aparecerá un cuadro de mensaje de error indicando que no se han seleccionado los registros a eliminar.	

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Conexión con BD perdida	Cuadro de dialogo que muestre la siguiente leyenda: “Ha ocurrido un error al cargar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador”.
E2	Error al eliminar los datos de la BD	Cuadro de dialogo que muestre la siguiente leyenda: "Ha ocurrido un error al eliminar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador".

A.1.7 Detalle del caso de uso “Modificar Artesanía”

Definición	
Caso de uso:	**Modificar Artesanía**

Tipo:	Básico.
Actor principal:	***Usuario***
Descripción:	Este caso de uso describe como el *usuario* puede modificar a una artesanía existente en la BD.
Pre condiciones:	El Usuario debe iniciar el sistema y localizarse en la ventana de artesanías.
Pos condiciones:	El Usuario podrá modificar a las artesanías existentes en la base de datos.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos				
Flujo básico:				
Modificar Artesanía				
Modificación de artesanía normal				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanías en el menú Local.	2.	El sistema muestra una ventana donde se muestran las artesanías que se encuentran en la Base de datos.	E1
3	El usuario opta por modificar una o varias artesanías de la base de datos y da clic en el menú Administrar→ Modificar.	4	Si existen datos cargados previamente en la lista mostrada, aparecerá una ventana mostrando la información de la artesanía.	E3
5.	Si el usuario ha decidido que la información es la correcta, actualiza el registro dando clic en el botón aceptar para seguir navegando por los registros y modificar las veces que se requiera.	6.	Si el usuario ha decidido terminar con la modificación de los registros da clic en el botón Aceptar para cerrar la ventana y mostrar los datos en la lista actualizada.	

Flujos alternativos:				
Modificación de artesanía con datos incorrectos				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanías en el menú Local.	2.	El sistema muestra una ventana donde se muestran las artesanías que se encuentran en la Base de datos.	E1
3.	El usuario opta por modificar una o varias artesanías de la base de datos y da clic en el menú Administrar→ Modificar o da enter con la tecla en un registro específico.	4.	Si existen datos cargados previamente en la lista mostrada, aparecerá una ventana mostrando la información del artesano.	E2
5.	Si el tipo de dato que el usuario ha insertado es correcta, y actualiza el registro dando clic en el botón aceptar, aparecerá un cuadro de mensaje indicando que los tipos de datos son incorrectos.		Si los datos fueron correctos, y el usuario ha decidido terminar con la modificación de los registros da clic en el botón Aceptar para cerrar la ventana y mostrar los datos en la lista actualizada.	

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Conexión con BD perdida	Cuadro de dialogo que muestre la siguiente leyenda: “Ha ocurrido un error al cargar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador”.

E2	Formato incorrecto de datos	Cuadro de dialogo que muestre la siguiente leyenda:"Ha ocurrido un error, verifique que el tipo de los datos sean los correctos o que no hayan campos vacíos".
E3	Error al modificar los datos de la BD	Cuadro de dialogo que muestre la siguiente leyenda: "Ha ocurrido un error al modificar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador".

A.1.8 Detalle de caso de uso "Búsqueda de Artesanías".

Definición	
Caso de uso:	**Búsqueda de Artesanías**
Tipo:	Básico.
Actor principal:	***Usuario***
Descripción:	Este caso de uso describe como el *usuario* puede buscar a una o varias artesanías existentes en la BD.
Pre condiciones:	El Usuario debe iniciar el sistema.
Pos condiciones:	El Usuario podrá buscar a una o varias artesanías.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos				
Flujo básico:				
Búsqueda de Artesanías				
Búsqueda de artesano con datos				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Artesanías en el menú Local.	2.	El sistema muestra una ventana donde se muestran las Artesanías que se encuentran en la Base de datos.	E1
3.	El usuario opta por buscar una o varias artesanías de la base de datos y da clic en el menú Mostrar→ Buscar.	4.	Se abre una ventana con un campo vacío para colocar el nombre completo o parte de el, para buscar la artesanía.	
5.	Si el usuario ha decidido que ha colocado la información correcta, y da clic en el botón buscar para iniciar el proceso de búsqueda de las artesanías por su nombre, con lo cuál se cerrará la ventana automáticamente y se podrán observar los datos en la ventana que la abrió.			

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Conexión con BD perdida	Cuadro de dialogo que muestre la siguiente leyenda: "Ha ocurrido un error al cargar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador".

A.1.9 Detalle de caso de uso "Navegación sobre clientes"

Definición	
Caso de uso:	**Navegación sobre clientes**
Tipo:	Básico.
Actor principal:	***Usuario***
Descripción:	Este caso de uso describe como el *usuario* puede navegar sobre los clientes que se hayan

	registrado en la página web de ArteOnline.
Pre condiciones:	El Usuario debe iniciar el sistema.
Pos condiciones:	El Usuario podrá observar a detalle los clientes registrados.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos				
Flujo básico:				
Navegación sobre Clientes.				
Navegación sobre clientes a través de la lista.				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Clientes en el menú Web.	2.	El sistema muestra una ventana donde se muestran los clientes que se hayan registrado en la Base de datos	E1
3.	El usuario opta por observar a detalle uno a uno a los clientes de la base de datos y da doble clic sobre el registro de la lista.	4.	Se abre una ventana donde se pueden observar los datos del cliente en varios campos. A partir del registro seleccionado, se podrá navegar hacia adelante o atrás para observar a los diferentes clientes.	
5.	Si el usuario ha decidido que ha visto la suficiente información da clic en el botón Cerrar para quitar la ventana.			

Flujos alternativos:				
Navegación sobre clientes a través del menú.				
Actor				
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Clientes en el menú Web.	2.	El sistema muestra una ventana donde se muestran los clientes que se hayan registrado en la Base de datos	E1
3.	El usuario opta por observar a detalle uno a uno a los clientes de la base de datos y da clic sobre la opción Detalles del menú Administración.	4.	Se abre una ventana donde se pueden observar los datos del cliente en varios campos. A partir del registro seleccionado, se podrá navegar hacia adelante o atrás para observar a los diferentes clientes.	
5.	Si el usuario ha decidido que ha visto la suficiente información da clic en el botón Cerrar para quitar la ventana.			

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Conexión con BD perdida	Cuadro de dialogo que muestre la siguiente leyenda: "Ha ocurrido un error al cargar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador".

A.1.10 Detalle de caso de uso "Navegación sobre órdenes de los clientes".

Definición	
Caso de uso:	**Navegación sobre órdenes de los clientes**
Tipo:	Básico.
Actor principal:	***Usuario***
Descripción:	Este caso de uso describe como el *usuario* puede navegar sobre las órdenes de los clientes que hayan realizado en la página web de ArteOnline.

Pre condiciones:	El Usuario debe iniciar el sistema y encontrarse en la ventana de clientes o en la ventana de órdenes.
Pos condiciones:	El Usuario podrá observar a detalle los clientes registrados.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos				
Flujo básico:				
Navegación sobre órdenes de los clientes				
Navegación sobre órdenes de los clientes desde la ventana de clientes.				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Clientes en el menú Web.	2.	El sistema muestra una ventana donde se muestran los clientes que se hayan registrado en la Base de datos	E1
3.	El usuario opta por observar las órdenes realizadas por un cliente en específico y selecciona un registro de la lista y da clic sobre la opción Ver orden.	4.	Se abre una ventana donde se pueden observar las órdenes del cliente.	
5.	Si el usuario opta por ver a detalle la orden, da doble clic sobre un registro de la lista o a través de la opción Detalle del menú Administración y aparecerá una ventana mostrado los datos de la orden.			

Flujos alternativos:				
Navegación sobre órdenes de los clientes desde la ventana de órdenes.				
Actor				
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Órdenes en el menú Web.	2.	El sistema muestra una ventana donde se muestran todas las órdenes de los clientes.	E1
3.	Si el usuario opta por ver a detalle una orden de cualquier cliente, da doble clic sobre un registro de la lista o a través de la opción Detalle del menú Administración y aparecerá una ventana mostrado los datos de la orden.			

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Conexión con BD perdida	Cuadro de dialogo que muestre la siguiente leyenda: "Ha ocurrido un error al cargar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador".

A.1.11 Detalle de caso de uso "Modificar status de la orden".

Definición	
Caso de uso:	**Modificar status de la orden**
Tipo:	Básico.
Actor principal:	***Usuario***
Descripción:	Este caso de uso describe como un usuario del sistema puede modificar el estado de la orden.
Pre condiciones:	El usuario debe estar en el sistema ArteOnLine.
Pos condiciones:	El usuario podrá modificar el estado de la orden, para que la artesanía llegue al cliente.
Elaborado por:	José Antonio Cruz Jiménez

Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos				
Flujo básico:				
Modificar status de la orden				
Modificación del status normal				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario se encuentra en la ventana de órdenes.	2.	El sistema muestra una ventana donde se observan las ordenes de los clientes que se encuentran en la Base de datos.	E1
3.	El usuario decide modificar el status de la orden para dar el seguimiento de la misma y da clic sobre el menú Administrar en la opción modificar.	4.	Si ha seleccionado algún elemento de la lista, aparecerá una ventana con las opciones de status de la orden.	E2
5.	Si el usuario ha decidido que ha escogido la opción correcta, da clic sobre el botón aceptar para actualizar la orden en la base de datos, cerrar de forma automática la ventana y actualizar el status del registro en la lista.			

Flujos alternativos:				
Modificación del status sin selección de registros				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario se encuentra en la ventana de órdenes.	2.	El sistema muestra una ventana donde se observan las ordenes de los clientes que se encuentran en la Base de datos.	E1
3.	El usuario decide modificar el status de la orden para dar el seguimiento de la misma y da clic sobre el menú Administrar en la opción modificar.	4.	Si no ha seleccionado algún elemento de la lista o si realizó una búsqueda previa sin resultados, aparecerá un mensaje indicando que no hay registro alguno a modificar.	

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Conexión con BD perdida	Cuadro de dialogo que muestre la siguiente leyenda: “Ha ocurrido un error al cargar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador”.
E2	Error al modificar los datos de la BD	Cuadro de dialogo que muestre la siguiente leyenda: "Ha ocurrido un error al actualizar el status de la orden, inténtelo más tarde. Si persiste el problema contáctese con el administrador".

A.1.12 Detalle caso de uso “Búsqueda de clientes”

Definición	
Caso de uso:	**Búsqueda de clientes**
Tipo:	Básico.
Actor principal:	***Usuario***
Descripción:	Este caso de uso describe como el *usuario* puede buscar a uno o varios clientes existentes en la BD.
Pre condiciones:	El Usuario debe iniciar el sistema y localizarse en la ventana de clientes.

Pos condiciones:	El Usuario podrá buscar a uno o varios clientes.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-06-2009
Fecha de actualización:	26-06-2009

Flujos				
Flujo básico:				
Búsqueda de clientes				
Búsqueda de clientes con datos				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Usuario decide ir a la ventana de Clientes en el menú Web.	2.	El sistema muestra una ventana donde se muestran los clientes que se encuentran en la Base de datos.	E1
3.	El usuario opta por buscar uno o varios clientes de la base de datos y da clic en el menú Mostrar→ Buscar.	4.	Se abre una ventana con un campo vacío para colocar el nombre completo o parte de el, para buscar al cliente.	
5.	Si el usuario ha decidido que ha colocado la información correcta, y da clic en el botón buscar para iniciar el proceso de búsqueda de los clientes por su nombre de usuario, con lo cuál se cerrará la ventana automáticamente y se podrán observar los datos en la ventana que la abrió.			

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Conexión con BD perdida	Cuadro de dialogo que muestre la siguiente leyenda: "Ha ocurrido un error al cargar los datos, inténtelo más tarde. Si persiste el problema contáctese con el administrador"

A.2 DETALLES DE CASOS DE USO APLICACIÓN ASP.NET.

A.2.1 Detalle caso de uso "Registrar Cliente"

Definición	
Caso de uso:	**Crear Cliente**
Tipo:	Básico.
Actor principal:	***Cliente Web***
Descripción:	Este caso de uso describe como un *cliente* puede registrarse en la BD para poder realizar su carro de compra y obtener los productos que se muestran en la página web de Arte On Line.
Pre condiciones:	El Usuario debe estar en la página web de Arte On Line.
Pos condiciones:	El Usuario podrá registrarse ante el sistema, autenticarse (login) y realizar ordenes.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos

Flujo básico:				
Registrar Cliente				
Registro de cliente normal				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web decide ir a la opción de registrarse en el menú Registro.	2.	El sistema muestra un formulario donde se muestran los campos vacios a llenar por parte del cliente.	E1
3.	Si el usuario ha insertado valores y opta por terminar el registro dando clic sobre el botón registro, automáticamente se redirigirá hacia la página de login, donde colocará su nombre de usuario y contraseña insertadas previamente.	4.		

Flujos alternativos:				
Registro de cliente sin datos.				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web decide ir a la opción de registrarse en el menú Registro.	2.	El sistema muestra un formulario donde se muestran los campos vacios a llenar por parte del cliente.	E1
3.	Si el usuario no ha insertado valores y opta por terminar el registro dando clic sobre el botón registro, aparecerán mensajes en los campos vacios indicando que deben colocarse valores.	4.		

Registro de cliente con contraseña no coincidentes.				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web decide ir a la opción de registrarse en el menú Registro	2.	El sistema muestra un formulario donde se muestran los campos vacios a llenar por parte del cliente.	E1
3.	Si el usuario ha insertado valores, pero las contraseñas que debió haber insertado anteriormente no coinciden y opta por terminar el registro dando clic sobre el botón registro, aparecerá un mensaje en los campos de password indicando que las contraseñas no coinciden.		.	
Registro de Cliente con nombre de usuario existente.				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web decide ir a la opción de registrarse en el menú Registro	2.	El sistema muestra un formulario donde se muestran los campos vacios a llenar por parte del cliente.	E1
3	Si el usuario ha insertado valores, pero el nombre de usuario que escribió ya existe y opta por terminar el registro dando clic sobre el botón			

	registro, aparecerá un mensaje en el campo de usuario indicando que el usuario ya existe.			

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Error al agregar al cliente	Texto que muestre la siguiente leyenda: "Ha ocurrido un error en el registro, inténtelo más tarde."

A.2.2 Detalle de caso de uso "Login de usuario"

Definición	
Caso de uso:	**Login de usuario**
Tipo:	Básico.
Actor principal:	***Cliente Web***
Descripción:	Este caso de uso describe como un *cliente* puede realizar el autenticarse en la página web de Arte On Line.
Pre condiciones:	El Usuario debe estar en la página de login de Arte On Line.
Pos condiciones:	El Usuario podrá realizar compras de los productos que se muestran en la página web Arte On Line, además de observar las órdenes realizadas.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos				
Flujo básico:				
Login de usuario				
Login de usuario con datos				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web se encuentra en la página de login de Arte On Line, en el que se muestran 2 campos vacíos donde colocará el nombre de usuario y el password proporcionados en el momento del registro.	2.	Si el usuario ha colocado datos sobre los dos únicos campos, da clic sobre el botón Login, automáticamente se redirigirá hacia la página de inicio, pero con la diferencia de poder realizar su carrito de compra y por ende su orden de compra.	E1

Flujos alternativos:				
Login de usuario sin datos				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web se encuentra en la página de login de Arte On Line, en el que se muestran 2 campos vacíos donde colocará el nombre de usuario y el password proporcionados en el momento del registro.	2.	Si el usuario no ha colocado datos sobre alguno de los dos únicos campos, da clic sobre el botón Login, aparecerá un mensaje en la parte inferior del campo que quedó vacío indicando que no puede quedar así.	

Login de usuario con nombre o password incorrectos

Actor		Sistema		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web se encuentra en la página de login de Arte On Line, en el que se muestran 2 campos vacíos donde colocará el nombre de usuario y el password proporcionados en el momento del registro.	2.	Si el usuario ha colocado datos sobre los dos únicos campos, da clic sobre el botón Login, pero los datos de nombre de usuario o contraseña son incorrectos aparecerá un mensaje de texto indicando que el nombre de usuario contraseña son incorrectos.	

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Error al cargar los datos.	Texto que muestre la siguiente leyenda: "Ha ocurrido un error al cargar los datos, inténtelo más tarde."

A.2.3 Detalle de caso de uso "Agregar ítems al carrito de compra" (Creación del carrito de compra)

Definición	
Caso de uso:	**Agregar ítems al carrito de compra**
Tipo:	Básico.
Actor principal:	***Cliente Web***
Descripción:	Este caso de uso describe como un *cliente* puede realizar el llenado del carrito de compra con los productos que se muestran en la página web de Arte On Line.
Pre condiciones:	El Usuario debe estar en la página web default de Arte On Line y estar logeado.
Pos condiciones:	El Usuario podrá agregar artesanías que se muestran en la página web Arte On Line hacia su carrito de compra y crear así su orden.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos				
Flujo básico:				
Agregar ítems al carrito de compra				
Agregar ítems al carrito de compra con usuario logeado.				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web se encuentra en la página de inicio con los datos de los productos. Decide agregar una artesanía hacia el carrito de compra y da clic sobre el texto "Agregar a carrito" que se encuentra en la parte final de cada producto.	2.	Si el usuario ha decidido que ha escogido las artesanías adecuadas siguiendo el paso anterior, da clic sobre la lista con el nombre "Carrito", para poder observar los productos y la cantidad empezando por 1.	E1
3	Si el cliente esta conforme con las cantidades que se colocaron pasa al siguiente caso de uso "Crear orden de compra"	4		

Flujos alternativos:		
Agregar ítems al carrito de compra con usuario no logeado.		
Actor	**Sistema**	

Paso	Acciones	Paso	Acciones	Excepción
1.	El Cliente Web se encuentra en la página de inicio con los datos de los productos. Decide agregar una artesanía hacia el carrito de compra y da clic sobre el texto "Agregar a carrito" que se encuentra en la parte final de cada producto.	2.	Si el usuario no se encuentra logeado aparecerá un cuadro de mensaje indicando que debe estar tanto registrado como haber iniciado sesión para poder acceder a esa función.	

Agregar ítems al carrito de compra con usuario logeado y cantidades superiores				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web se encuentra en la página de inicio con los datos de los productos. Decide agregar una artesanía hacia el carrito de compra y da clic sobre el texto "Agregar a carrito" que se encuentra en la parte final de cada producto.	2.	Si el usuario ha decidido que ha escogido las artesanías adecuadas siguiendo el paso anterior, da clic sobre la lista con el nombre "Carrito", para poder observar los productos y la cantidad empezando por 1.	
3	Si el cliente no está conforme con las cantidades que se colocaron, coloca la cantidad que requiere sobre el cuadro de texto en el que se muestra la cantidad a pedir.		Si la cantidad que insertó supera a la cantidad de artículos disponibles en la base de datos, da clic sobre el texto "Modificar" para actualizar al carrito de compra con lo que los nuevos valores se conservarán para dar paso al caso de uso "Crear orden de compra"	

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Error al agregar ítem en el carrito de compra.	Texto que muestra la siguiente leyenda: "El ítem no se ha podido agregar , inténtelo más tarde"

A.2.4 Detalle de caso de uso "Crear orden de compra"

Definición	
Caso de uso:	**Crear orden de compra.**
Tipo:	Básico.
Actor principal:	***Cliente Web***
Descripción:	Este caso de uso describe como un *cliente* puede crear su orden de compra con los productos que colocó en su carrito de compra a través de la página web de Arte On Line.
Pre condiciones:	El Usuario debe estar en la página web default de Arte On Line, estar logeado y tener ítems en su carrito de compra.
Pos condiciones:	El Usuario podrá adquirir las artesanías que se muestran en la página web Arte On Line y obtener el número de orden de esa venta.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos	
Flujo básico:	
Crear orden de compra	
Creación de la orden de compra sin errores.	

Actor		Sistema		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web se encuentra en la página de carrito con los datos de los productos listos. Decide crear la orden de compra y da clic sobre el botón Finalizar.	2.	Si no existen errores en la creación de la orden se muestra un texto de mensaje indicando que la orden fue creada exitosamente.	E1
Flujos alternativos:				
Creación de la orden de compra con errores.				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web se encuentra en la página de carrito con los datos de los productos listos. Decide crear la orden de compra y da clic sobre el botón Finalizar.	2.	Si existen errores en la creación de la orden se muestra un texto de mensaje indicando que hubo un error en la creación de la orden.	E1

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Error al crear la orden	Texto que muestre la siguiente leyenda: "Ocurrió un error al crear la orden, inténtelo más tarde y verifique sus datos"

A.2.5 Detalle de caso de uso "Seguimiento de órdenes"

Definición	
Caso de uso:	**Seguimiento de órdenes.**
Tipo:	Básico.
Actor principal:	***Cliente Web***
Descripción:	Este caso de uso describe como un *cliente* puede observar las órdenes de compra que realizó a través de la página web de Arte On Line.
Pre condiciones:	El Usuario debe estar logeado y estar en la página de carrito de compra.
Pos condiciones:	El Usuario podrá observar las órdenes que ha realizado en la página web Arte On Line, incluyendo las artesanías, cantidades y precios.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos				
Flujo básico:				
Seguimiento de órdenes				
Seguimiento de ordenes normal				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web se encuentra en la página de carrito con o sin los datos de los productos listos. Decide ver las órdenes de compra que ha realizado y da clic sobre el botón seguimiento de órdenes.	2.	Se muestra un formulario en donde se pueden observar las órdenes de compra realizadas por el cliente.	E1

Flujos alternativos:				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web se encuentra en la página	2.	Se muestra un formulario en donde se pueden	E1

	de carrito con o sin los datos de los productos listos. Decide ver las órdenes de compra que ha realizado y da clic sobre el botón seguimiento de órdenes.		observar las órdenes de compra realizadas por el cliente. SI no existen órdenes realizadas solo se muestra un texto indicando que no existen órdenes.	

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Conexión con BD perdida	Texto que muestre la siguiente leyenda: "Error al cargar los datos, intentelo más tarde."

A.2.6 Detalle de caso de uso "Búsqueda de artesanías"

Definición	
Caso de uso:	**Búsqueda de artesanías**
Tipo:	Básico.
Actor principal:	***Cliente Web***
Descripción:	Este caso de uso describe como un *cliente* puede buscar artesanías por su nombre través de la página web de inicio de Arte On Line.
Pre condiciones:	El Usuario debe estar en la página de inicio del sistema.
Pos condiciones:	El Usuario podrá observar las artesanías en la página web Arte On Line de acuerdo al nombre o parte de él.
Elaborado por:	José Antonio Cruz Jiménez
Versión:	0.0.1
Fecha de creación:	26-05-2009
Fecha de actualización:	26-05-2009

Flujos				
Flujo básico:				
Búsqueda de artesanías				
Búsqueda de artesanías normal				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web se encuentra en cualquier página del sistema. Decide ver las artesanías de acuerdo a algún nombre de la artesanía o parte de el, así que escribe sobre el campo Buscar y da clic sobre el botón buscar.	2.	Automáticamente se redirige al usuario hacia la página de inicio donde se pueden observar las artesanías de acuerdo a los parámetros que introdujo en el campo de búsqueda.	E1

Flujos alternativos:				
Búsqueda de artesanías sin datos				
Actor		**Sistema**		
Paso	**Acciones**	**Paso**	**Acciones**	**Excepción**
1.	El Cliente Web se encuentra en cualquier página del sistema. Decide ver las artesanías de acuerdo a algún nombre de la artesanía o parte de el, así que da clic sobre el botón buscar.	2.	Si el usuario no escribió algún valor sobre el campo, no se realizará búsqueda alguna.	E1

Excepciones		
Identificador	**Nombre**	**Respuesta del Sistema**
E1	Error al cargar los datos.	Texto que muestre la siguiente leyenda: "Ocurrió un error al cargar los datos de las artesanías, inténtelo de nuevo."

ANEXO B.

Esquema de definición haciendo uso del lenguaje de definición de datos (*DML*) para *SQL Server* para la base de datos ArteOnLine.

B.1 Tabla para artesanos:

```
CREATE TABLE [dbo].[Artesano](
	[IDArtesano] [int] IDENTITY(1,1) NOT NULL,
	[Nombre] [nvarchar](50) NOT NULL,
	[Apellidos] [nvarchar](50) NOT NULL,
	[Calle] [nvarchar](50) NOT NULL,
	[Ciudad] [nvarchar](30) NOT NULL,
	[Estado] [nvarchar](20) NOT NULL,
	[Telefono] [nvarchar](30) NULL,
	[Email] [nvarchar](50) NULL,
	[Pseudonimo] [nvarchar](30) NULL,
	[FechaRegistro] [datetime] NOT NULL,
	[FechaModificacion] [datetime] NULL,
	[Imagen] [varbinary](max) NULL,
 CONSTRAINT [PK_Artesanos] PRIMARY KEY CLUSTERED
(
	[IDArtesano] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]

) ON [PRIMARY];
```

B.2 Tabla para las artesanías:

```
CREATE TABLE [dbo].[Artesania](
	[IDArtesania] [int] IDENTITY(1,1) NOT NULL,
	[IDArtesano] [int] NOT NULL,
	[DescripcionArtesania] [nvarchar](150) NOT NULL,
	[Cantidad] [int] NOT NULL,
	[Precio] [decimal](10, 2) NOT NULL,
	[IDClasificacion] [int] NOT NULL,
	[Imagen] [varbinary](max) NULL,
	[FechaAlta] [datetime] NOT NULL,
	[FechaModificacion] [datetime] NULL,
 CONSTRAINT [PK_Artesanias] PRIMARY KEY CLUSTERED
(
	[IDArtesania] ASC
 )WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
 ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY],

CONSTRAINT [FK Artesanias IDArtesano] FOREIGN KEY([IDArtesano])
REFERENCES [dbo].[Artesano] ([IDArtesano]),

CONSTRAINT [FK_Artesanias_IDClasificacion] FOREIGN KEY([IDClasificacion])
REFERENCES [dbo].[ClasificacionArtesania] ([IDClasificacion])
ON UPDATE CASCADE ON DELETE CASCADE

) ON [PRIMARY];
```

B.3 Tabla para la clasificación de las artesanías:

```
CREATE TABLE [dbo].[ClasificacionArtesania](
	[IDClasificacion] [int] IDENTITY(1,1) NOT NULL,
	[Clasificacion] [nvarchar](30) NOT NULL,
	[DescripcionClasificacion] [nvarchar](600) NOT NULL,
	[FechaModificacion] [datetime] NULL,
 CONSTRAINT [PK_ClasificacionArtesanias] PRIMARY KEY CLUSTERED
(
```

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

B.4 Tabla para los clientes:

```
CREATE TABLE [dbo].[Cliente](
	[IDCliente] [int] IDENTITY(1,1) NOT NULL,
	[Nombre] [nvarchar](100) NOT NULL,
	[Apellidos] [nvarchar](50) NULL,
	[NombreUsuario] [nvarchar](50) NOT NULL,
	[PasswordUsuario] [nvarchar](150) NOT NULL,
	[Calle] [nvarchar](50) NOT NULL,
	[Ciudad] [nvarchar](30) NOT NULL,
	[Estado] [nvarchar](20) NOT NULL,
	[Telefono] [nvarchar](30) NULL,
	[Email] [nvarchar](50) NULL,
	[FechaRegistro] [datetime] NOT NULL,
	[FechaModificacion] [datetime] NULL,
	[IDTipoCliente] [int] NOT NULL,
 CONSTRAINT [PK_Clientes] PRIMARY KEY CLUSTERED
(
	[IDCliente] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
 ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]

CONSTRAINT [FK Clientes IDTipoCliente] FOREIGN KEY([IDTipoCliente])
REFERENCES [dbo].[TipoCliente] ([IDTipoCliente]) ON UPDATE CASCADE ON DELETE CASCADE

) ON [PRIMARY];
```

B.5 Tabla para el tipo de cliente:

```
CREATE TABLE [dbo].[TipoCliente](
	[IDTipoCliente] [int] IDENTITY(1,1) NOT NULL,
	[Tipo] [nvarchar](20) NOT NULL,
 CONSTRAINT [PK_TipoClientes] PRIMARY KEY CLUSTERED
(
	[IDTipoCliente] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY];
```

B.6 Tabla para las órdenes:

```
CREATE TABLE [dbo].[Orden](
	[IDPedido] [int] IDENTITY(1,1) NOT NULL,
	[FechaOrden] [datetime] NOT NULL,
	[FechaEntrega] [datetime] NOT NULL,
	[IDCliente] [int] NOT NULL,
	[IDStatusOrden] [int] NOT NULL,
 CONSTRAINT [PK_OrdenPedido] PRIMARY KEY CLUSTERED
(
	[IDPedido] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY],

CONSTRAINT [FK_Orden_Cliente] FOREIGN KEY([IDCliente])
REFERENCES [dbo].[Cliente] ([IDCliente]) ON UPDATE CASCADE ON DELETE CASCADE,

CONSTRAINT [FK_Orden_StatusOrden] FOREIGN KEY([IDStatusOrden])
REFERENCES [dbo].[StatusOrden] ([IDStatusOrden]) ON UPDATE CASCADE ON DELETE CASCADE

) ON [PRIMARY];
```

B.7 Tabla DetalleOrden:

```
CREATE TABLE [dbo].[DetalleOrden](
	[IDDetallePedido] [int] IDENTITY(1,1) NOT NULL,
	[IDPedido] [int] NOT NULL,
	[IDArtesania] [int] NOT NULL,
	[PrecioUnitario] [decimal](12, 2) NOT NULL,
	[Cantidad] [int] NOT NULL,
	[FechaModificacion] [datetime] NULL,
 CONSTRAINT [PK_DetallePedidos] PRIMARY KEY CLUSTERED
(
	[IDDetallePedido] ASC
 )WITH (PAD INDEX  = OFF, STATISTICS NORECOMPUTE  = OFF, IGNORE DUP KEY = OFF,
 ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY],

CONSTRAINT [FK_DetalleOrden_Orden] FOREIGN KEY([IDPedido])
REFERENCES [dbo].[Orden] ([IDPedido]) ON UPDATE CASCADE ON DELETE CASCADE,

CONSTRAINT [FK_DetallePedidos_IDArtesania] FOREIGN KEY([IDArtesania])
REFERENCES [dbo].[Artesania] ([IDArtesania])

) ON [PRIMARY];
```

B.8 Tabla StatusOrden:

```
CREATE TABLE [dbo].[StatusOrden](
	[IDStatusOrden] [int] IDENTITY(1,1) NOT NULL,
	[Status] [nvarchar](50) NOT NULL,
 CONSTRAINT [PK StatusOrden] PRIMARY KEY CLUSTERED
(
	[IDStatusOrden] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW ROW LOCKS  = ON, ALLOW PAGE LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY];
```