UNIVERSIDAD ESTATAL A DISTANCIA
VICERRECTORÍA ACADÉMICA
ESCUELA DE CIENCIAS EXACTAS Y NATURALES

DIPLOMADO EN INFORMÁTICA

GUÍA DE ESTUDIO

PROGRAMACIÓN AVANZADA

CÓDIGO 830

Elaborada por

Carlos H. Hernández Alvarado

Edición académica

Virginia Ramírez

Encargado de cátedra

Roberto Morales

Revisión filológica

Oscar Alvarado

Esta guía de estudio ha sido confeccionada para ser utilizada en el curso Programación Avanzada código 830 del programa Diplomado en Informática que imparte la UNED.

Presentación

La programación es una de las áreas más importantes en las ciencias de la computación e informática. Es un campo que se encuentra en auge y cambio permanente, y que tiene la característica de crear un profesional integral, pues lo relaciona de manera cercana con los objetivos y procesos de cada uno de los ámbitos donde se desenvuelve. Es por ello que el estudio de técnicas avanzadas de programación, como las que se desarrollan en esta guía, son de vital trascendencia para los estudiantes de esta carrera, ya que les permitirán transformar dichos objetivos y procesos en *software* especializado de una manera más eficaz y eficiente.

La presente guía de estudio se basa en la segunda edición del libro *C#: Cómo Programar,* de Harvey M. Deitel y Paul J. Deitel, que además de introducir al lector en uno de los lenguajes de programación más modernos y versátiles, proporciona herramientas, técnicas y ejercicios avanzados que permiten comprender los conceptos más complejos tanto de la programación en general, como del lenguaje específico y la forma efectiva de ponerlos en práctica.

Cada uno de los temas desarrollados contiene una serie de ejercicios y ejemplos que serán esenciales para la comprensión de los conceptos.

Para que el alumno vea facilitado su accionar e interacción con esta guía, el libro de texto anexa un *CD-ROM* con los capítulos del libro en formato digital, así como los archivos de los ejercicios que en él se desarrollan. Adicionalmente, deberá contar preferiblemente con una conexión a Internet, con el fin de que pueda consultar el material adicional y complementario al que se hace mención en el transcurso del presente documento.

Ayudas para facilitar el uso del texto y de la guía de estudio

Con el fin de facilitar la lectura de este documento y hacerlo más amigable, el texto incluye algunos íconos, que identifican secciones especiales, remarcables o importantes y que le ayudarán a identificar los ejercicios, ideas, precauciones, recomendaciones y enlaces a otros materiales, para comprender y desarrollar de una manera más ágil la materia del curso.

El ícono del bombillo remarca ideas y recomendaciones importantes que se desarrollan tanto en el libro de texto como en la guía de estudio. Estas le permitirán resolver con más facilidad los ejercicios planteados en los distintos capítulos y en los laboratorios que ha de realizar durante el avance del curso.

El ícono del libro señala los enlaces a materiales adicionales de estudio, tales como: referencias a otros libros, enlaces de Internet o artículos de relevancia.

El ícono del atleta identifica los ejercicios de autoevaluación, que reforzarán sus conocimientos al final de cada subtema. Las soluciones a estos ejercicios y los proyectos de *Visual C#* se encuentran al final de cada sección o disponibles en la plataforma virtual.

El ícono de tubo de ensayo identifica los ejercicios que se realizarán al final de los apartados y que permitirán poner en práctica los conceptos aprendidos. Los laboratorios normalmente consisten en dos ejercicios: uno que se plantea y se resuelve en este documento y otro en la tutoría correspondiente.

El ícono de precaución advierte sobre posibles errores comunes que se pueden encontrar tanto en el desarrollo de las actividades como de los laboratorios, con el fin de que se puedan identificar y evitar a priori.

Contenido

I. Generalidades del curso y de la guía de estudio

Objetivos generales

El estudiante será capaz de:

1. Utilizar las características, las herramientas, las particularidades y los conceptos avanzados del lenguaje de programación orientado a objetos C#.

2. Programar aplicaciones complejas en el lenguaje C#, usando los conceptos de subprocesamiento múltiple, bases de datos y redes.

3. Manipular los elementos que componen la interfaz gráfica de las aplicaciones en C#.

Objetivos específicos

Durante el desarrollo del curso, el estudiante:

1. Identifica los elementos principales del lenguaje de programación orientado a objetos C#.

2. Conoce técnicas de la programación orientada a objetos, en el lenguaje de programación C#.

3. Utiliza el manejo de excepciones y errores en el lenguaje de programación C#.

4. Elabora aplicaciones utilizando los conceptos de interfaz gráfica para los usuarios en el lenguaje de programación C#.

5. Maneja el concepto de subprocesamiento múltiple y lo aplica en la programación de sus aplicaciones en el lenguaje de programación C#.

6. Utiliza diferentes consultas a bases de datos a través del lenguaje de programación C#.

7. Manipula los diferentes objetos y conceptos del lenguaje de programación involucrados en la utilización de los protocolos de transmisión de datos y datagramas.

8. Utiliza las colecciones para agrupar datos en el lenguaje de programación C#.

Requisitos y recomendaciones para el curso

Este curso tiene como fin brindar herramientas complejas en un lenguaje moderno, poderoso y sencillo. Es necesario que el estudiante posea conocimientos previos en programación orientada a objetos y lenguajes avanzados de alto nivel como *Java* o C++.

Se recomienda haber finalizado las materias 831: Introducción a la programación, 824: Programación intermedia y 826: Bases de datos, así como cursos avanzados de lectura en inglés, dado que muchas de las recomendaciones y lecturas complementarias se encuentran en este idioma.

Adicionalmente, es requisito indispensable contar con los recursos computacionales adecuados, de forma que pueda instalar y ejecutar las herramientas de programación que se utilizarán en el curso. Las tutorías se brindan en aquellos centros universitarios que posean laboratorio de cómputo y según el número de estudiantes matriculados.

Software por utilizar en el curso

Para el desarrollo de los laboratorios y de las aplicaciones que se programarán en el curso, son necesarias las herramientas *Express* de Microsoft, que poseen una licencia gratuita.

Para descargar las herramientas de *Express* de Microsoft, diríjase a las siguientes direcciones en Internet: *Microsoft Visual Studio Express.*
<http://www.microsoft.com/express/download/>.
Microsoft SQL Server Express 2008 with Tools.
<http://www.microsoft.com/express/sql/download/default.aspx>.

Puede descargar las ayudas complementarias de los ambientes de desarrollo, que poseen una licencia gratuita. Estos recursos le ayudarán a solventar dudas específicas del lenguaje o sus características. Si lo desea, baje la librería *MSDN Express* y *MSDN Full* de la siguiente dirección:
<http://www.microsoft.com/express/download/msdn/Default.aspx>.

Estructura de la guía de estudio

La guía del presente curso consta de 8 temas del libro de texto que se han distribuido en 4 partes o secciones principales, que corresponden con las 4 tutorías, distribuidos de la forma siguiente.

1. Introducción al lenguaje orientado a objetos C#. Capítulos 3, 4 y 5.
2. Particularidades del lenguaje C#. Capítulos 6 y 8.
3. Manejo de excepciones en C#. Capítulo 12.
4. Conceptos de interfaz gráfica. Capítulos 13 y 14.
5. Subprocesamiento múltiple. Capítulo 15.
6. Bases de datos y componentes *ADO .NET.* Capítulo 20.
7. Redes: *Sockets* basados en flujos y datagramas. Capítulo 23.
8. Colecciones. Capítulo 26.

Secciones de la guía

Primera sección: Generalidades del lenguaje e interfaz gráfica (Primera tutoría)

Tema 1: Introducción al lenguaje orientado a objetos C#

3.3 – Cómo crear una aplicación simple en *Visual C# Express*

4.2 – Clases, objetos, métodos, propiedades y variables de instancia

5.2 – Estructuras de control

Ejercicios de autoevaluación

Tema 2: Particularidades del lenguaje C#

6.8 – Operadores lógicos

8.2 – Arreglos

8.3 – Declaración y creación de arreglos

8.4 – Ejemplos acerca del uso de los arreglos

Ejercicios de autoevaluación

Tema 3: Manejo de excepciones en C#

12.2 – Generalidades acerca del manejo de excepciones

12.3 – Ejemplo: División entre cero sin manejo de excepciones

12.4 - Ejemplo: manejo de las excepciones *DividebyZeroException y FormatException*

Ejercicios de autoevaluación

Tema 4: Conceptos de interfaz gráfica

13.2 – Formularios Windows

13.3 – Manejo de eventos

13.4 – Propiedades y distribución de los controles

13.5 – Controles *Label, TextBox y Button*

13.6 – Controles *GroupBox* y Panel

13.7 – Controles *CheckBox y RadioButton*

13.9 – Controles *ToolTip*

13.10 – Control *NumericUpDown*

13.11 – Manejo de los eventos del ratón

13.12 – Manejo de los eventos del teclado

14.2 – Menús

14.3 – Control *MonthCalendar*

14.4 – Control *DateTimePicker*

14.6 – Control *ListBox*

14.7 – Control *CheckedListBox*

14.8 – Control *ComboBox*

14.9 – Control *TreeView*

14.10 – Control *ListView*

14.11 – Control *TabControl*

14.12 – Ventanas de la interfaz de múltiples documentos (*MDI*)

Ejercicios de autoevaluación

Laboratorios

Segunda sección: Subprocesamiento múltiple (Segunda tutoría)

Tema 5: Subprocesamiento múltiple

15. 1 – Introducción

15.2 – Estados de los subprocesos: ciclo de vida de un subproceso

15.3 – Prioridades y programación de subprocesos

15.4 – Creación y ejecución de subprocesos

15.5 – Sincronización de subprocesos y la clase Monitor

15.9 – Subprocesamiento múltiple con *GUIs*

Sincronización de procesos con semáforos

Ejercicios de autoevaluación

Laboratorios

Tercera sección: Bases de datos y colecciones (Tercera tutoría)

Tema 6: Bases de datos y componentes *ADO .NET*

20.1 – Introducción

20.2 – Bases de datos relacionales

20.5 – Modelo de objetos *ADO .NET*

20.6 – Programación con *ADO .NET*: extraer información de una base de datos

20.9 – Uso de un objeto *DataSet* para leer y escribir *XML*

Ejercicios de autoevaluación

Tema 8: Colecciones

26.2 – Generalidades acerca de las colecciones

26.4 – Colecciones no genéricas

26.5 – Colecciones genéricas

26.6 – Colecciones sincronizadas

Ejercicios de autoevaluación

Laboratorios

Cuarta sección: Redes (Cuarta tutoría)

Tema 7: Redes. *Sockets* basados en flujos y datagramas

23.1 – Introducción

23.2 – Comparación entre la comunicación orientada a la conexión y la comunicación sin conexión

23.3 – Protocolos para transportar datos

23.4 – Establecimiento de un servidor *TCP* simple

23.5 – Establecimiento de un cliente *TCP* simple

23.6 – Interacción entre cliente/servidor mediante *sockets* de flujo

23.7 – Interacción entre cliente/servidor sin conexión mediante datagramas

23.8 – Juego de Tres en Raya cliente/servidor mediante el uso de un servidor con subprocesamiento múltiple

Ejercicios de autoevaluación

Laboratorios

Laboratorios

Al finalizar cada sección, se encuentran los laboratorios. Estos son ejercicios que presentan una complejidad más alta y que abarcan toda la materia estudiada. Usted deberá realizar dos por sección. Uno se soluciona en esta guía, con el fin de facilitar su comprensión; el otro se presenta para ser resuelto en su casa, o en las tutorías del curso.

Elabore siempre un reporte de laboratorio en un documento digital. Como cualquier ingeniero, lleve apuntes de sus experimentos. Mantenga un record de las principales funciones, instrucciones, librerías y utilidades con las que resolvió los ejercicios. Repasar esquemas o reportes, lo cual le ayudará a estudiar la materia del curso. Recuerde que en el examen debe "programar en papel", por lo que mantener un resumen se vuelve primordial.

II. Primera sección (Primera tutoría) Generalidades del lenguaje e interfaz gráfica

Esta sección es introductoria y pretende acercar al estudiante al nuevo lenguaje, así como aprender a manipular el *IDE* de desarrollo, para crear aplicaciones eficientes, agradables y eficaces. Lea y estudie las secciones de los temas 1, 2, 3 y 4 que se han destacado en la estructura de la guía. Cuando haya finalizado su lectura, se le recomienda elaborar un cuadro comparativo a manera de resumen, con los principales conceptos aprendidos, y compararlos con los que conoce de los lenguajes y cursos anteriores.

Si usted ha llevado el curso 824 – Programación Intermedia, utilizando el lenguaje de programación *JAVA*, lea el "*C# from Java Orange Book*", de Rob Miles. Es un libro de 25 páginas que brinda prácticas a los programadores para hacer la transición desde *JAVA* hacia C# más transparente y sencilla. Puede obtenerlo de manera gratuita desde el siguiente enlace, o descargarlo desde la plataforma virtual:
<http://www.robmiles.com/c-yellow-book/C%20Sharp%20from%20Java%20Orange%20Book%202009.pdf>.

Si usted ya tiene conocimiento en el lenguaje C#, se recomiendan los siguientes libros como fuentes de consulta adicionales, los cuales puede descargar desde la plataforma virtual:

- Rob Miles: *C# Yellow Book. 2008.*
- José Antonio González Seco: *El lenguaje de programación C#. 2006.*
- Eric Buttow y Tommy Ryan: "*C#: Your visual blueprint for building .NET Applications*".

Tema 1: Introducción al lenguaje orientado a objetos C#

En el libro los ejemplos programados y pantallas vienen realizados con el ambiente de *Visual C# 2005*, el *IDE* no posee cambios drásticos, y utilizar la versión 2008 no tiene un mayor impacto.

Recuerde que con este *IDE* se pueden crear aplicaciones en consola y en ventanas para Windows, similares a las que se usan diariamente. El *IDE* tiene la característica de poder personalizarse a gusto del programador, así que, una vez que haya instalado el softwar*e* de *Visual C# Express* 2008, personalícelo con las recomendaciones que se brindan en la sección 3.3 del libro, página 53.

Familiarícese con otras características y objetos destacables del ambiente de desarrollo, como el *Intellisense*, las ventanas de Explorador de soluciones, las propiedades de los objetos, la lista de errores, los íconos, los mensajes informativos para compilar y ejecutar una aplicación y los tipos de errores que se presentan en el *IDE* al realizar estas acciones.

Este es un curso de programación avanzada y se asume que usted conoce las nociones relacionadas con este tema. Como repaso, se le recomienda resumir las definiciones de clase, objeto, método, atributos, herencia y polimorfismo. Se incluyen las figuras 1 y 2 con el fin de refrescarlos y se utilizan en el ejemplo 1. Asuma que se trata de una bicicleta e identifique los conceptos de métodos y atributos.

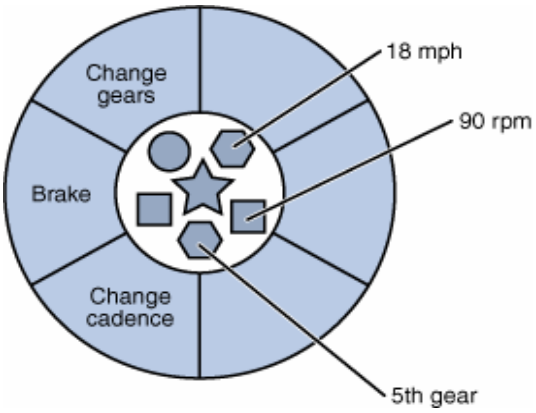

Figura 1: Un objeto bicicleta.
Fuente: <http://java.sun.com/docs/books/tutorial/java/concepts/object.html>.

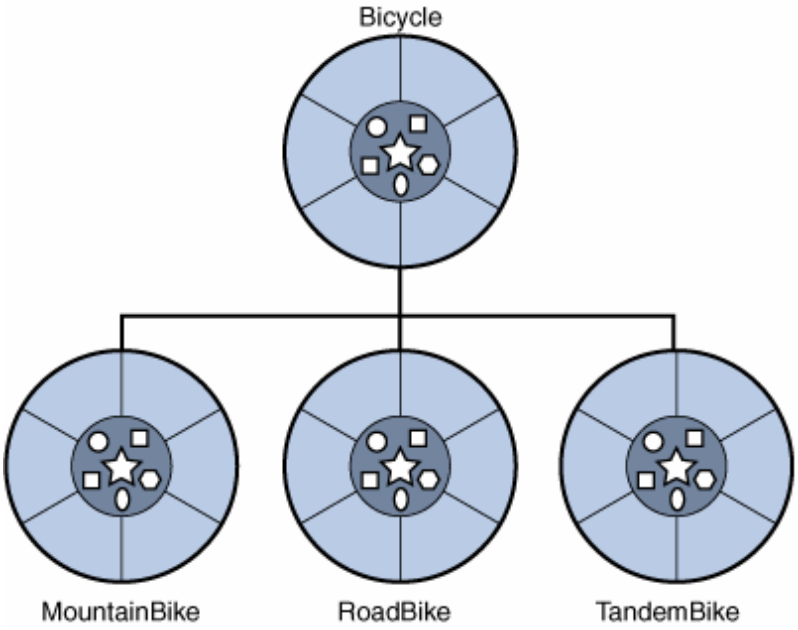

Figura 2: Herencia entre objetos.
Fuente: <http://java.sun.com/docs/books/tutorial/java/concepts/inheritance.html>.

¿Cómo se codifica un objeto como el anterior en el lenguaje C#?

Para adentrarse en el lenguaje, se inicia con el ejemplo 1 de cómo puede escribirse el anterior objeto en código C#. Observe y señale los renglones más relevantes.

Ejemplo 1

```
class Bicicleta{
  protected int Marcha;
  protected int Pedaleo;
  protected double Velocidad;

  public void cambiarMarcha(int cambio){
    Marcha = cambio;
  }//fin de método cambiarMarcha

  public void pedalearBicicleta (int rpm){
    Pedaleo = rpm;
  }//fin de Método PedalearBicicleta
  public int calcularVelocidad (){
    Velocidad = (Pedaleo * Marcha)/100;
  }// fin de método calcular Velocidad

}//end class Radio
```

Preste atención a las principales características del ejemplo 1. La clase consta de 3 atributos y 3 métodos que los manipulan. Una instancia de la clase mostrada es un objeto.

Para finalizar el primer tema de la guía, es importante asimilar las diferentes estructuras de control que posee el lenguaje C#. Para facilitar dicho aprendizaje, se brinda el cuadro 1.1, que muestra las distintas estructuras de control presentes en el lenguaje *JAVA* y el lenguaje C#, como continuidad entre el curso 824: Programación Intermedia y este. Si desea conocer con más detalle cada una, refiérase a los capítulos 5 y 6 del libro de texto, páginas 115 y 143.

Cuadro 1.1
Estructuras de control en lenguajes de programación *JAVA* y C#

Estructura de control en *JAVA*	**Estructura de Control en C#**
if (condición)	*if* (condición)
If (condición verdadera)… *else*	*if*(condición verdadera)… *else*
while (condición verdadera)	*while* (condición verdadera)
for	*for*
switch	*switch*
break, continue	*break, continue*

 ¡Tenga cuidado con el problema del "*else* suelto". El compilador de C# siempre asocia un *else* con el *if* por el cual está precedido, a menos de que se le indique lo contrario con el uso de llaves "{ }". Evite la mala asociación utilizándolas siempre, aunque la estructura de control vaya seguida solamente por una instrucción, como se observa en el ejemplo 2.

Ejemplo 2

```
bool flagCheck = true;
if (flagCheck == true){
   Console.WriteLine("The flag is set to true.");
}
else{
   Console.WriteLine("The flag is set to false.");
}
```

Ejercicios de autoevaluación 1.1

Utilizando el *Visual C# Express*, realice la siguiente codificación.

1. Escriba una pequeña clase en código C# que represente a un avión. Debe tener 5 atributos que representan: el estado actual del avión (volando, aterrizando, despegando, en tierra) la posición de los alerones, la posición del tren de aterrizaje, la posición de las puertas del avión (abiertas o cerradas) y la potencia de los motores.

2. La clase debe tener cuatro métodos: Descargar, aterrizar, volar y despegar. Estas funciones alteran los atributos según sea necesario.

Tema 2: Particularidades del lenguaje C#

Al igual que las estructuras de control, C# contiene los mismos operadores lógicos que *JAVA*. Obsérvelos en el cuadro 1.2.

Cuadro 1.2
Operadores lógicos del lenguaje de programación C#

Operador	Función
AND Condicional - **&&**	Compara dos o más expresiones. Retorna verdadero si y sólo si todas las expresiones son verdaderas.
OR Condicional - **||**	Compara dos o más expresiones. Retorna verdadero cuando alguna de las expresiones es verdadera.
AND Lógico Booleano - **&**	Compara dos o más expresiones con el mismo efecto del *AND Condicional*. Siempre evalúa las dos expresiones.
OR Lógico Booleano - **|**	Compara dos o más expresiones con el mismo efecto del *OR Condicional*. Siempre evalúa las dos expresiones.
OR Exclusivo Lógico Booleano – **^**	Compara dos o más expresiones. Retorna verdadero cuando una de las expresiones es verdadera y la otra es falsa.
Operador Lógico de Negación - **!**	Permite invertir el significado o valor de verdad de una expresión.

Arreglos en C#

El manejo de arreglos en el lenguaje de programación C#, es un poco distinto a otros. Con el fin de aprenderlo mejor, utilice el material del *CD-ROM*. Descargue el archivo *cap08.zip* del disco compacto y descomprima su contenido. En el *folder* cap08, abra los ejemplos según el concepto que quiera repasar. Los ejemplos 3 y 4 explican dos figuras del libro que realizan un ejercicio de manipulación de arreglos. Preste atención al código y a los detalles relevantes que se mencionan en ellos.

Ejemplo 3

Figura 8.2 página 219

```
// Fig. 8.2: InicArreglo.cs
// Creación de un arreglo.
using System;

public class InicArreglo
{
   public static void Main( string[] args )
   {
      int[] arreglo; // declara un arreglo llamado arreglo

      // crea el espacio para el arreglo y lo inicializa con ceros
predeterminados
      arreglo = new int[ 10 ]; // 10 elementos int

      Console.WriteLine( "{0}{1,8}", "Índice", "Valor" ); // encabezados

      // imprime en pantalla el valor de cada elemento del arreglo
      for ( int contador = 0; contador < arreglo.Length; contador++ )
         Console.WriteLine( "{0,5}{1,8}", contador, arreglo[ contador ] );
   } // fin de Main
} // fin de la clase InicArreglo
```

Detalles relevantes de la figura 8.2

- Nótese la colocación de los paréntesis cuadrados en la declaración del arreglo.
- La dimensión del arreglo se especifica hasta su inicialización.
- La propiedad *Length* de un arreglo especifica el número de elementos.

Ejemplo 4

Figura 8.8 página 225

```
// Fig. 8.8: EncuestaEstudiantil.cs
// Aplicación para analizar encuestas.
using System;

public class EncuestaEstudiantil
{
   public static void Main(string[] args)
```

```
   {
      // arreglo de respuestas a la encuesta
      int[] respuestas = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6,
         10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6, 5, 6, 7, 5, 6,
         4, 8, 6, 8, 10 };
      int[] frecuencia = new int[11]; // arreglo de contadores de
frecuencia

      // para cada respuesta, selecciona el elemento de respuestas y usa
ese valor
      // como subíndice de frecuencia para determinar el elemento a
incrementar
      for (int respuesta = 0; respuesta < respuestas.Length; respuesta++)
         ++frecuencia[respuestas[respuesta]];

      Console.WriteLine("{0}{1,11}", "Calificación", "Frecuencia");

      // imprime en pantalla el valor de cada elemento del arreglo
      for (int calificacion = 1; calificacion < frecuencia.Length;
calificacion++)
         Console.WriteLine("{0,12}{1,11}", calificacion,
frecuencia[calificacion]);
   } // fin de Main
} // fin de la clase EncuestaEstudiantil
```

Detalles relevantes de la figura 8.8

- Nótese las distintas formas de declarar e inicializar un arreglo.
- El operador de incremento suma uno al campo respectivo según la aparición de un valor determinado.
- La función *Main ()* recibe como parámetro un arreglo de argumentos.
- Utilización de la palabra reservada *new* para realizar la asignación de memoria nueva para el arreglo.

Como ejercicio complementario, realice el mismo análisis para las figuras 8.6 y 8.7 del libro de texto, páginas 222 y 224.

Ejercicios de autoevaluación 1.2

1. Utilizando el código de los ejercicios de autoevaluación 1.1, programe la lógica para que su avión valide su estado antes de cambiarlo, con el fin de que siga un orden lógico.
    a. Aclaración: Un avión no debería poder volar si no ha despegado, o no debería poder descargar si está volando, etc.

2. Debe programar dicha lógica dentro de un método de la clase.

Tema 3: Manejo de excepciones en C#

El manejo de excepciones es un tema complejo que debe ser repasado con detalle cuando se conoce un nuevo lenguaje de programación. Un uso correcto de ellas permite hacer un programa más robusto, aumentando la tolerancia a fallas de cualquier pieza de *software*.

Práctica programada 1

Con el fin de refrescar los conocimientos en el manejo de excepciones se recomienda descargar del *CD-ROM* el archivo *cap12.zip* y descomprimirlo en su disco duro. Analice los ejemplos de las figuras 12.1 página 387 y 12.2 página 390 y realice las actividades 1 y 2 respectivamente.

Actividad 1

Abra el archivo. Siga los pasos de ejecución que se presentan en la figura 12.1 del libro y responda las siguientes preguntas.

- ¿Cuáles son las excepciones que se lanzan en este ejemplo?
- ¿Por qué aparecen dichas excepciones?
- ¿Cómo se llaman las líneas que se despliegan cuando ocurre la excepción?
- ¿Puede un programa seguir en ejecución cuando se ha producido una excepción no controlada?

Actividad 2

Abra el archivo. Siga los pasos de ejecución que se presentan en la figura 12.2 del libro y responda las siguientes preguntas.

- ¿Cuáles son las instrucciones que se utilizan para el manejo de excepciones?

- Investigue sobre el bloque *finally*. ¿Cuál es su función dentro del manejo de excepciones?
- ¿Puede el manejo de excepciones simplificarse de alguna forma en el ejemplo? ¿De qué forma?

A manera de resumen, el cuadro 1.3 contiene algunas excepciones que se lanzan en el lenguaje de programación C#:

Cuadro 1.3
Excepciones comunes del lenguaje de programación C#

Excepción	Ocurrencia
System.ArithmeticException	Clase base para excepciones que ocurren durante la ejecución de operaciones aritméticas, tales como: *System.DivideByZeroException* *System.OverflowException*
System.ArrayTypeMismatchException	Se lanza cuando se intenta guardar un elemento en un arreglo y hay un fallo debido a que el tipo de datos del elemento que se intenta guardar es incompatible con el tipo de datos del arreglo.
System.DivideByZeroException	Lanzada cuando se intenta realizar una división de un número entre cero.
System.IndexOutOfRangeException	Se lanza cuando se intenta indexar un arreglo en una posición que no existe.
System.InvalidCastException	Lanzada cuando se intenta realizar una conversión de un tipo de dato a otro que no es compatible.
System.OutOfMemoryException	Se lanza cuando una asignación de memoria falla.
System.StackOverflowException	Lanzada cuando la pila de llamados de funciones es demasiado grande, seña de una recursión infinita.


Si desea ampliar sus conocimientos sobre el manejo de excepciones en el lenguaje C#, puede consultar el siguiente enlace. Adicionalmente puede descargar el documento "*Comparativa de Manejo de Excepciones*" de Francisco Ortín Soler, desde la plataforma *Web* del curso <http://msdn.microsoft.com/es-es/library/ms173160(VS.80).aspx>.

Es una excelente práctica incluir el manejo de excepciones por defecto en las aplicaciones, aunque las mismas no sean las versiones finales.

Entre los errores comunes de programación se encuentra dejar al usuario "a ciegas" cuando ocurre una excepción. Considere que, aunque se controlen, sus programas deben ser lo suficientemente estables y amigables como para continuar su ejecución luego de un error y, adicionalmente, informar al usuario que este se ha producido.

Tema 4: Conceptos de interfaz gráfica

Una interfaz gráfica diseñada de manera adecuada permite a los usuarios aprender la nueva aplicación más rápidamente y vencer de forma más fácil su resistencia al cambio. Es, sin duda alguna, la carta de presentación de sus aplicaciones.

Los formularios Windows son las ventanas donde se ubican los elementos de una *GUI*. Contienen controles y componentes que pueden tener o no una representación gráfica, como por ejemplo un botón o un componente *Timer*. Se adjunta en esta guía la figura 13.3, página 414 del libro de texto, con el fin de que marque los controles que no conoce e investigue sobre ellos en los apartados posteriores.

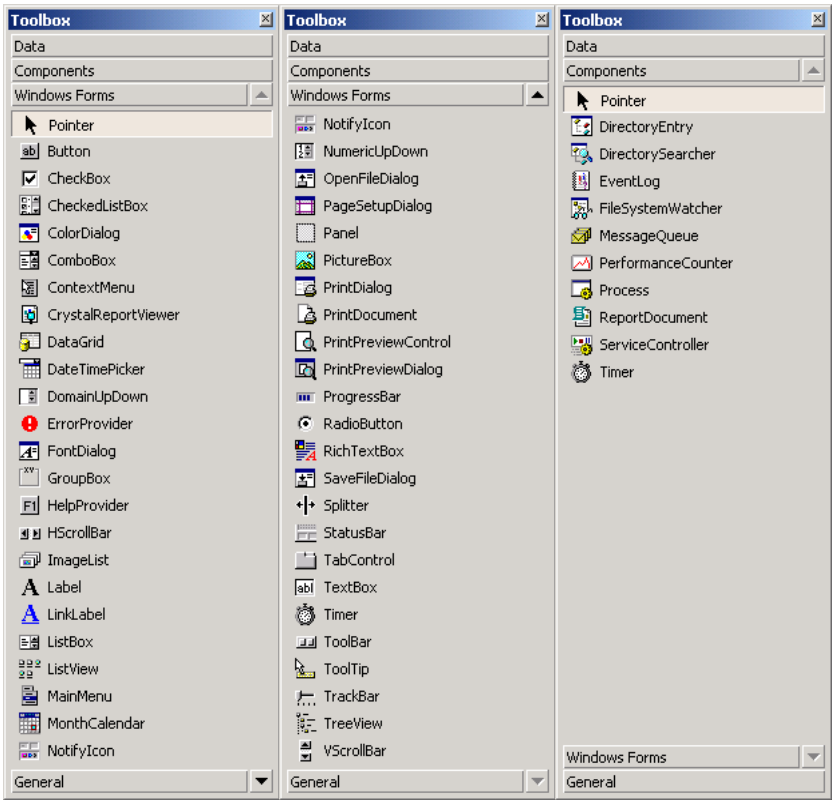

Figura 3: Controles comunes de los formularios Windows.
Fuente: C#. Cómo programar.

Manejo de eventos en la interfaz gráfica

Un evento es una acción que se ejecuta sobre una interfaz gráfica de una aplicación. Básicamente es una orden que el usuario realiza y por la cual espera una respuesta.

Las aplicaciones modernas están diseñadas para actuar y efectuar funciones según las necesidades y órdenes de los usuarios, lo que es lo mismo, son controladas por eventos.

Los controles de una interfaz poseen muchas propiedades que pueden ser cambiadas en tiempo de diseño o de ejecución. Observe la figura 13.8 del libro. En la ventana de propiedades del *IDE* de *Visual C#* pueden verse, así como algunos de los eventos. A continuación, se adjunta la ventana de propiedades de *Visual C# 2008*.

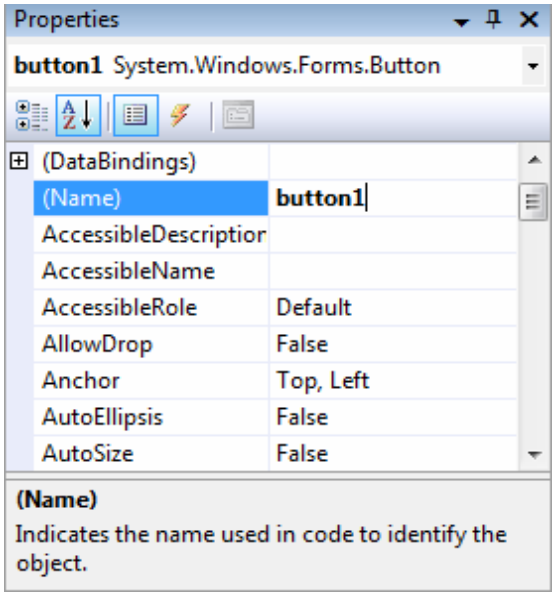

Figura 4: Ventana de propiedades de objetos de *Visual C# 2008*.

Como práctica para conocer los eventos de un objeto o control, realice el ejercicio 1.

Ejercicio 1

- Abra el *Visual C# 2008* y cree una aplicación Windows *Form*, como la de la figura 5.

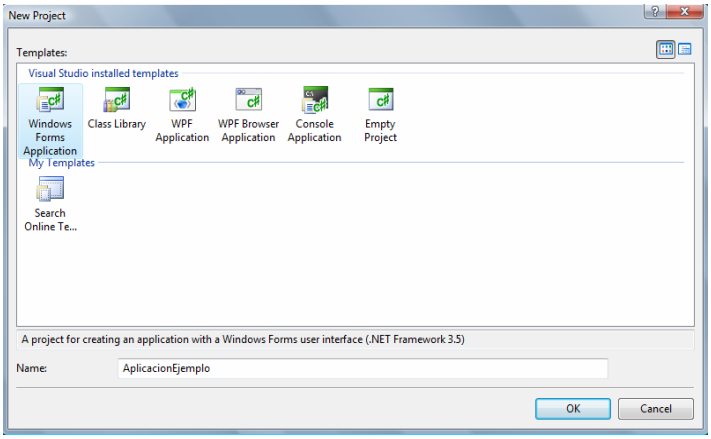

Figura 5: Ventana de nuevo proyecto en *Visual C# Express*.

- En el formulario principal, inserte un control de tipo *Button* y otro de tipo *ComboBox*, como se muestra en la figura 6.

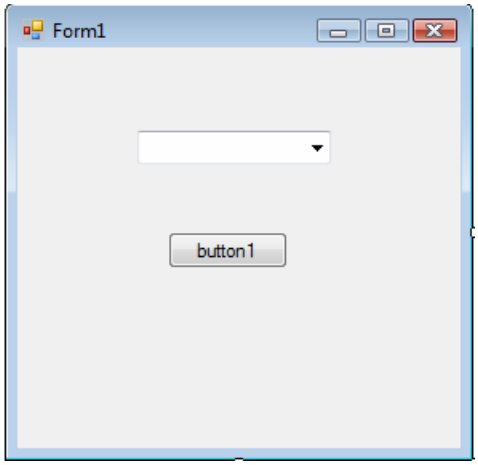

Figura 6: Un formulario de Windows con dos controles.

- Seleccione el control de tipo *Button* y, en la ventana de propiedades, elija los eventos del botón, como se observa en la figura 7.

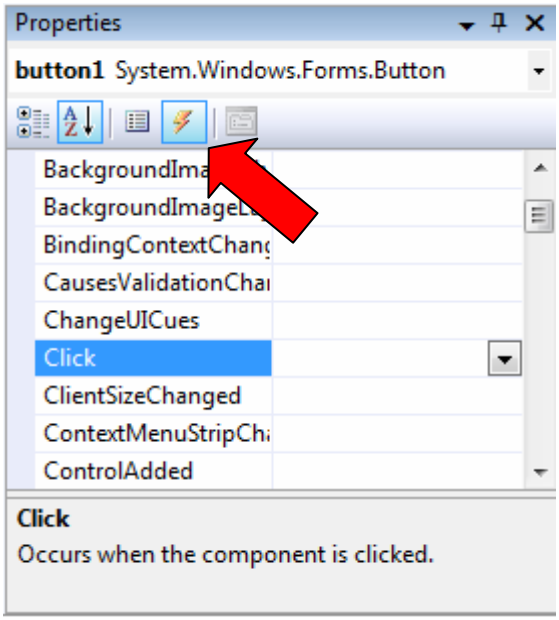

Figura 7: Propiedades del control *Button*.

- Algunos de los eventos encontrados en el botón son los siguientes:
    - *Click:* Ocurre cuando se hace *clic* sobre el componente.
    - *MouseHover:* Ocurre cuando el *mouse* se encuentra sobre el componente, pero sin presionarlo.
    - *Resize:* Ocurre cuando el componente cambia de tamaño.
    - *Leave:* Ocurre cuando el componente pierde el foco, o sea, cuando no es el control activo de la ventana.

- Haga doble *clic* sobre el evento que desea programar y de forma automática el ambiente de desarrollo creará la función relacionada con el evento del componente dentro del formulario. Usted deberá codificar las acciones relacionadas con el evento, como se muestra en la figura 8.

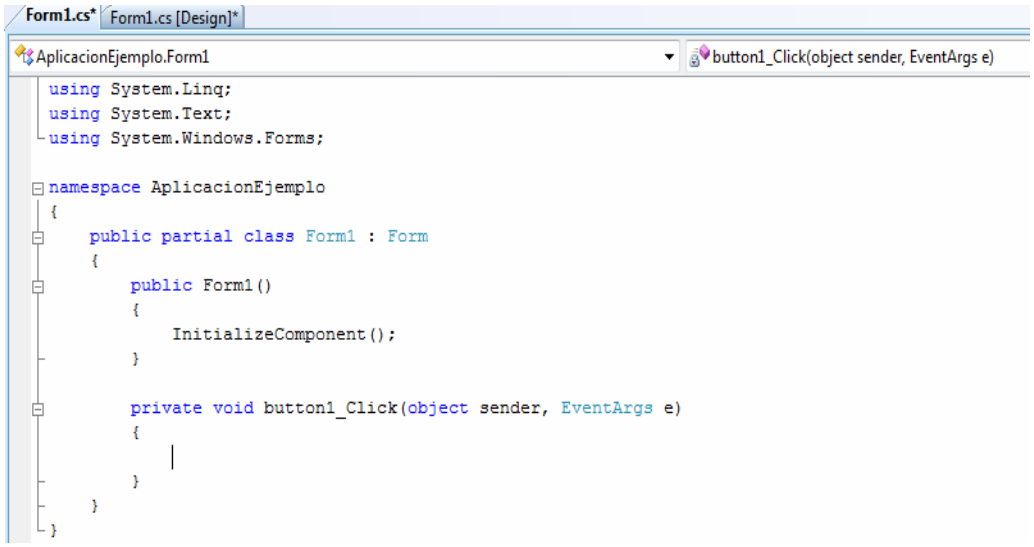

Figura 8: Código del evento *clic* de un control tipo *Button.*

Ahora usted deberá realizar el mismo proceso con otros dos componentes y documentarlos en el cuadro 4.

Cuadro 4
Eventos para los controles de interfaz gráfica

Componente	Evento	¿Cuándo sucede el evento?

Controles de interfaz gráfica

Como ha podido notar en las secciones anteriores de esta guía, el ambiente gráfico de C# posee una enorme variedad de controles que le permiten ejecutar y programar de una manera rápida y eficiente gran cantidad de operaciones. Debe leer todos los apartados del capítulo 13 relacionados con controles (13.5 a 13.10), así como los del capítulo 14 que se relacionan con controles de *GUI* (14.3 a 14.11), de forma tal que pueda utilizarlos en el futuro para crear aplicaciones complejas en pocos pasos.

Conocer las funciones, eventos y propiedades de los distintos controles de interfaz gráfica es una de las principales habilidades que un programador avanzado debe poseer. También se obliga a tener la pericia suficiente para investigar sobre las distintas funciones de los componentes que brinda el lenguaje, ya que toma menos tiempo el comprender el funcionamiento de uno e implementarlo, que crear una solución desde cero.

A manera de resumen, y como ayuda para sus labores de creación de interfaces y aplicaciones, se le presenta el cuadro 5 con los principales controles que se mencionan en

el libro, así como una pequeña explicación de su funcionalidad. Para ver una descripción más detallada, refiérase al libro de texto, capítulos 3 y 4.

Cuadro 5
Controles de interfaz de usuario comunes

Nombre del Control	Descripción
Label	Proporcionar información de texto que el usuario no puede modificar directamente.
TextBox	Mostrar texto o permitir el ingreso de texto a través del teclado.
Button	Control que oprime el usuario para desencadenar una acción específica.
CheckBox	Cuadro que está en blanco o contiene una marca de verificación.
RadioButton	Botón de opción con dos estados. Seleccionado y no seleccionado.
NumericUpDown	Control que restringe el rango de entrada de un usuario a un número específico de opciones.
DateTimePicker	Permite seleccionar una fecha. Evita el problema del formato.
ListBox	Permite al usuario ver y seleccionar uno o varios elementos de una lista.
CheckedListBox	Derivado de la clase *ListBox*. Incluye un *checkbox* en seguida de cada uno de los elementos de la lista.
ComboBox	Lista desplegable donde solo puede ser seleccionado un valor.
TreeView	Muestra nodos de forma jerárquica en un árbol.
ListView	Similar al *Listbox*, pero permite mostrar íconos enseguida de los elementos de la lista.
TabControl	Crea ventanas con cejillas. Permite especificar más información en el mismo espacio de un formulario.

Ventanas de interfaz de múltiples documentos

Lea la sección 14.12 página 494 del libro de texto y clasifique a continuación las siguientes aplicaciones de Windows en *SDI* y *MDI y* colóquelas en el cuadro 6.

- *Office Word*
- *Paint*
- *Notepad*
- *Internet Explorer 6.0*
- *Wordpad*
- *Visual C# 2008*
- *Internet Explorer 7*
- *Windows Media Player*

Cuadro 6
Clasificación de aplicaciones *Multiple Document Interface* y *Single Document Interface*

MDI	***SDI***

Algunas consideraciones sobre *MDI* y *SDI*

- ¿Qué sucede cuando un usuario abre un archivo que está relacionado con una aplicación *MDI*?

    - Cuando un usuario abre un archivo que se relaciona con una aplicación *MDI*, se crea una nueva clase hija (*Form* Hijo) dentro de la clase *MDI* Padre y ahí se visualiza el nuevo archivo.
- ¿Qué sucede si hace lo mismo con una aplicación *SDI*?
    - Cuando un usuario abre un archivo relacionado con una aplicación *SDI*, el archivo se abre en una nueva instancia del programa, en un formulario completamente aparte.


Cuando pueda utilizar *MDI*, úselo como primer recurso. Las aplicaciones *MDI* son más ordenadas, más fáciles de comprender y facilitan el manejo interno de *Forms*. Adicionalmente son usables y accesibles.

Ejercicio de investigación adicional

Con el fin de fomentar el hábito de la investigación en el alumno, que es una de las destrezas avanzadas que todo ingeniero en informática debe poseer, se le propone realizar una sobre dos temas relacionados con las interfaces gráficas.

En el diseño de interfaces gráficas se involucran dos conceptos muy relevantes: la usabilidad y la accesibilidad. Indague en Internet acerca de ambos y realice un resumen de no más de 2 páginas con toda la información. Adjúntela a su tarea corta #1. Debe contestar las siguientes preguntas.

1. Según estándares internacionales, ¿cuál es el concepto de usabilidad?
2. ¿Cuáles son los principios básicos de la usabilidad?
3. Mencione 3 beneficios que se obtienen al elaborar *software* usable.
4. ¿Qué es un *software* accesible?
5. ¿Qué significan las siglas *WCAG* y en qué consisten?
6. ¿Qué son los niveles de conformidad de accesibilidad?

Piense siempre en todos los mercados meta. Elabore sus aplicaciones utilizando siempre los estándares de usabilidad y accesibilidad. Es más sencillo programar una aplicación que sea usable y accesible desde el inicio, que modificarla cuando ya ha sido programada.

Ejercicios de autoevaluación 1.3

Con base en el ejemplo de la clase avión, desarrolle lo siguiente:

1. Programe una interfaz gráfica sencilla que permita controlar tres aviones.

2. La interfaz debe contar con tres *RadioButtons* que representan a los aviones.

3. Coloque cuatro botones con las acciones de aterrizar, despegar, volar y descargar. Programe cada uno de estos eventos para los dos aviones.

4. Utilice la lógica programada en el ejercicio de autoevaluación 1.2 para validar que las acciones que se solicitan a un avión tengan un orden coherente.

5. Su aplicación debe avisar al usuario con una pantalla emergente si está tratando de realizar una operación prohibida sobre el avión.

Laboratorios

Resuelva los laboratorios 1.1 y 1.2 aplicando los conocimientos aprendidos en esta sección de la guía de estudio.

Laboratorio 1.1

Se anexan algunas pantallas con el fin de que pueda seguir el ejercicio y resolverlo.

Realice los siguientes pasos.

1. Abra el *Visual C# Express 2008* y cree una nueva aplicación de Windows *Forms,* como la de la figura 9.

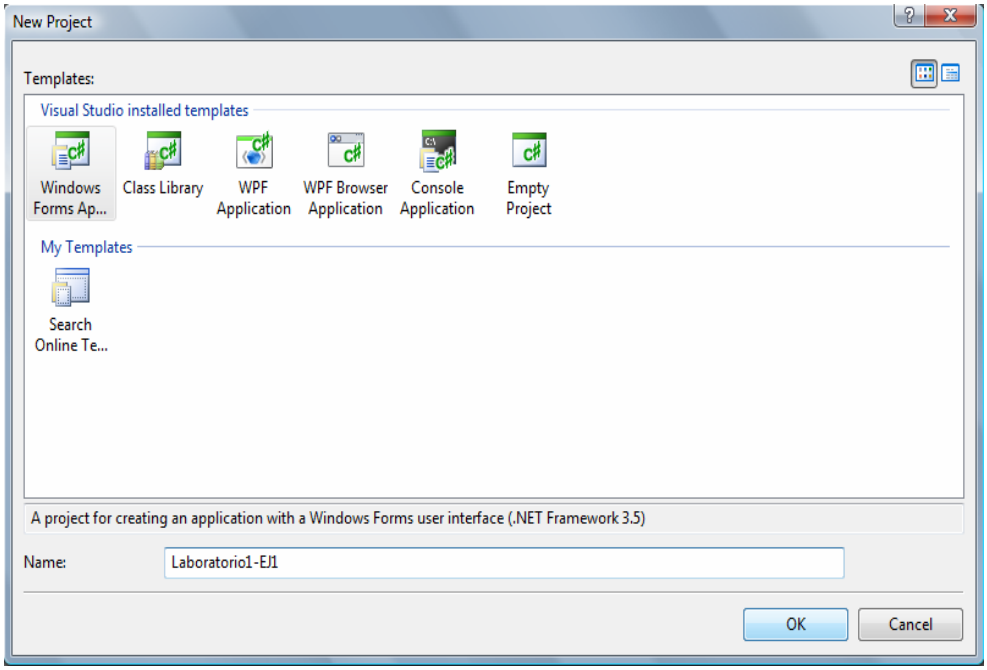

Figura 9: Ventana de nuevo proyecto.

2. Renombre el formulario principal como “Ejercicio1”, como en la figura 10.

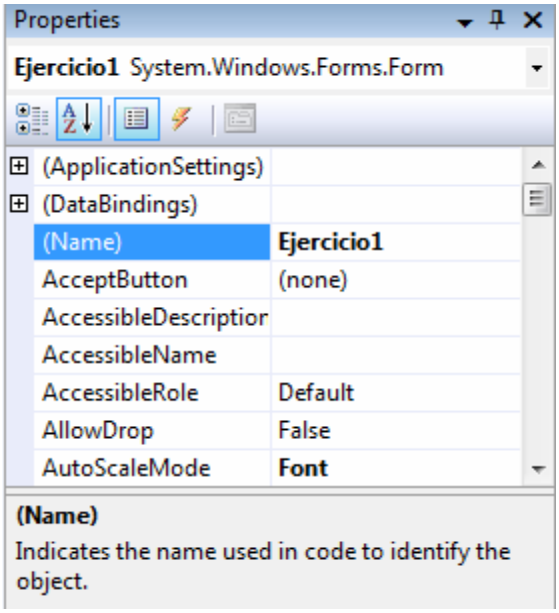

Figura 10: Propiedades del formulario principal.

3. Haga más grande el formulario de forma tal que pueda colocar otros dentro de él, similar al de la figura 11.

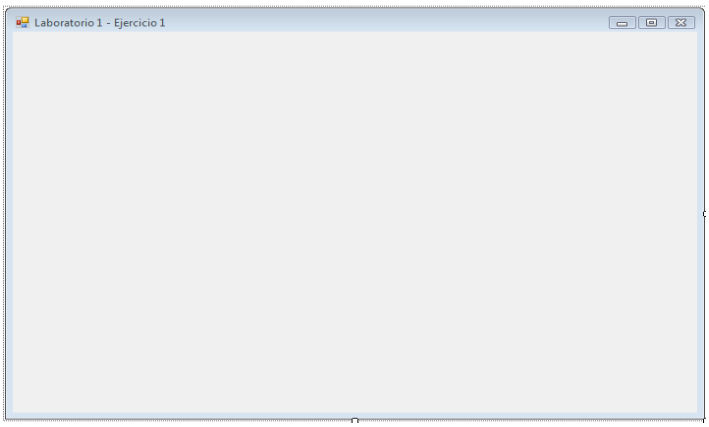

Figura 11: Formulario principal de la aplicación.

4. Haga que el formulario "Ejercicio1" sea un formulario *MDI* padre, como el de la figura 12.

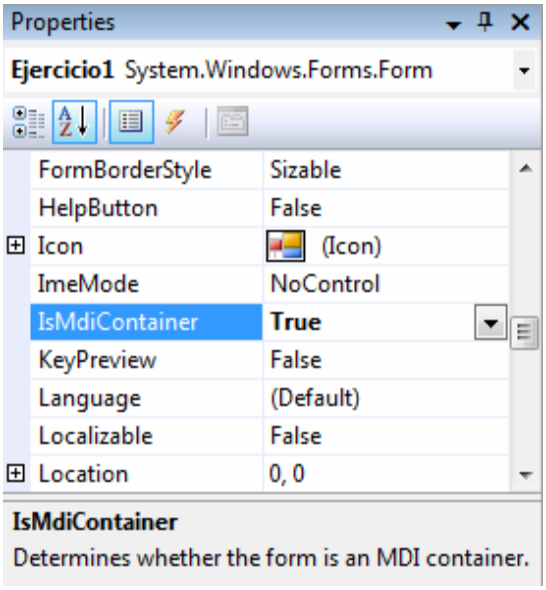

Figura 12: Propiedad *MDI* del formulario principal.

5. Agregue un nuevo objeto formulario al proyecto, como se observa en la figura 13, haciendo *clic* derecho con el *mouse* sobre el ícono del proyecto.

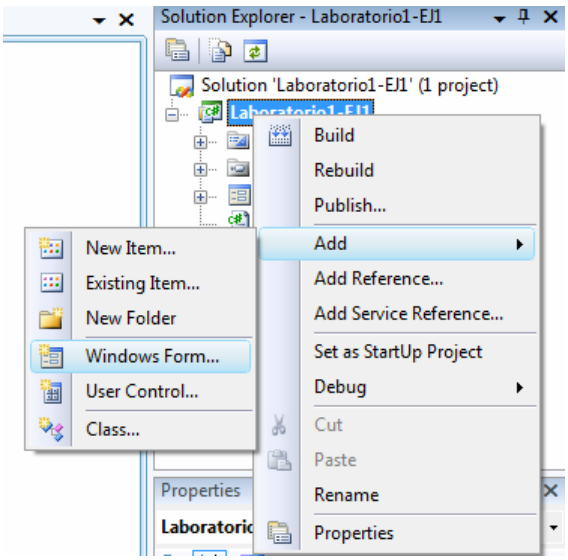

Figura 13: Agregar un formulario a la aplicación.

6. Renombre el formulario como “Ejemplo1”, como en la figura 14.

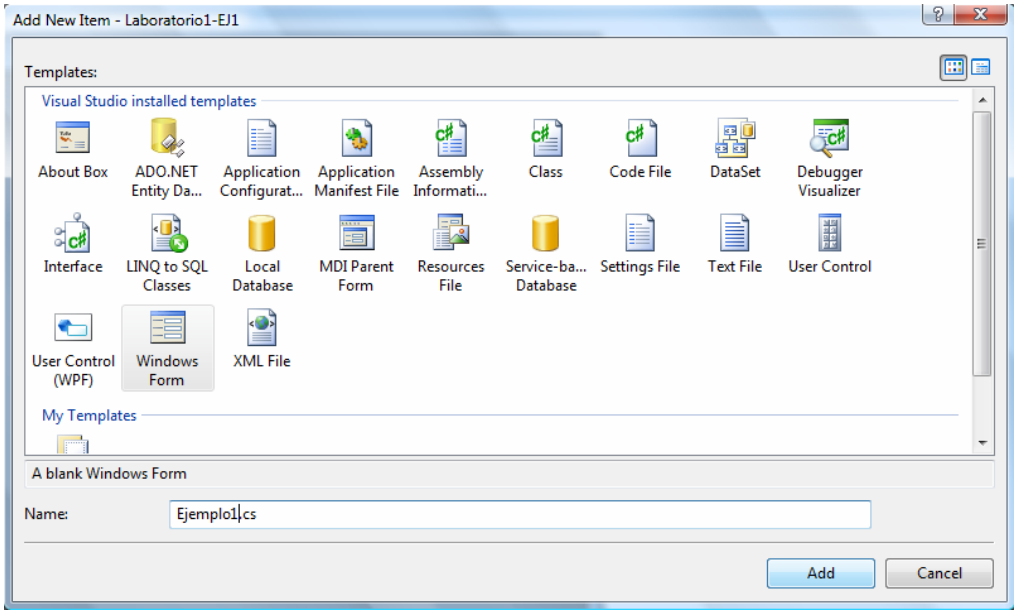

Figura 14: Ventana de nuevo formulario.

7. Seleccione nuevamente el formulario *MDI* padre. En la barra de herramientas, agregue un objeto *MenuStrip* al formulario principal, como se observa en la figura 15.

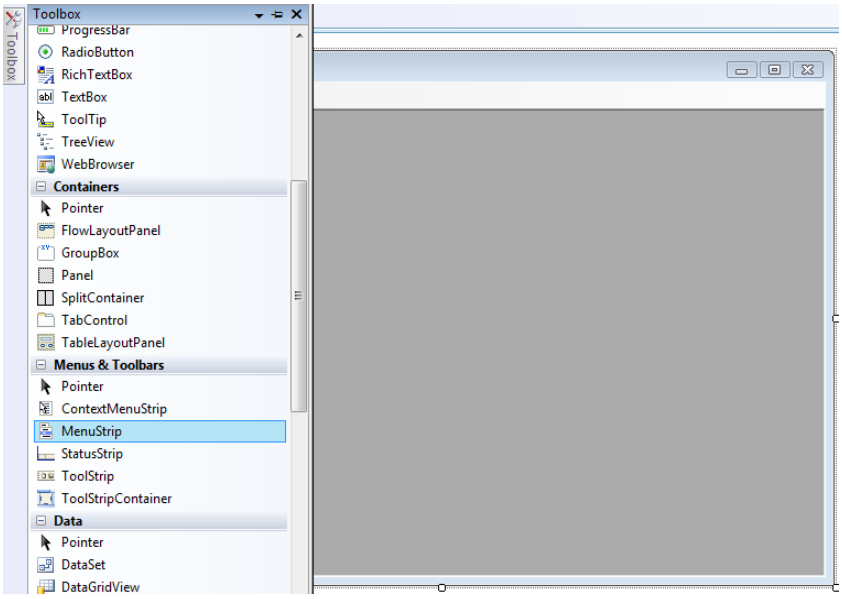

Figura 15: Barra de herramientas de formulario.

8. Cree un menú haciendo doble *clic* en los espacios designados. Nombre los espacios como "Ejemplos" y sus hijos como "Ejemplo 1", como en la figura 16 y así de forma sucesiva.

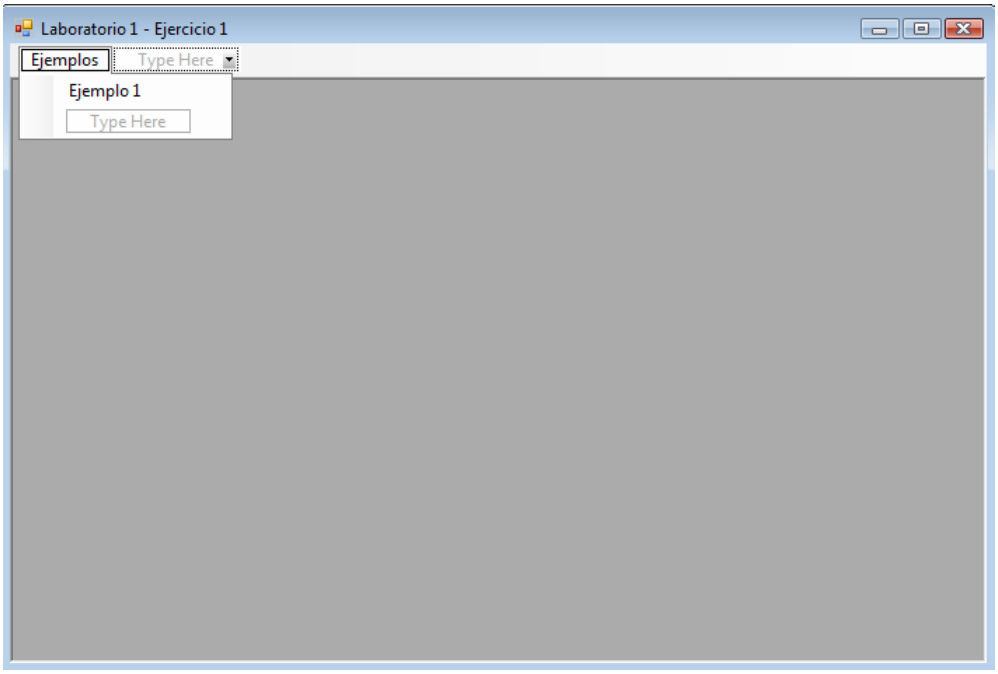

Figura 16: Objeto *MenuStrip*.

9. Haga doble *clic* sobre el primer elemento del menú ("Ejemplo 1"). Esto abrirá el editor de código. Copie las siguientes líneas en el editor como se observa en la figura 17.

```
Ejemplo1 frmEx1 = new Ejemplo1();
frmEx1.MdiParent = this;
frmEx1.Show();
```

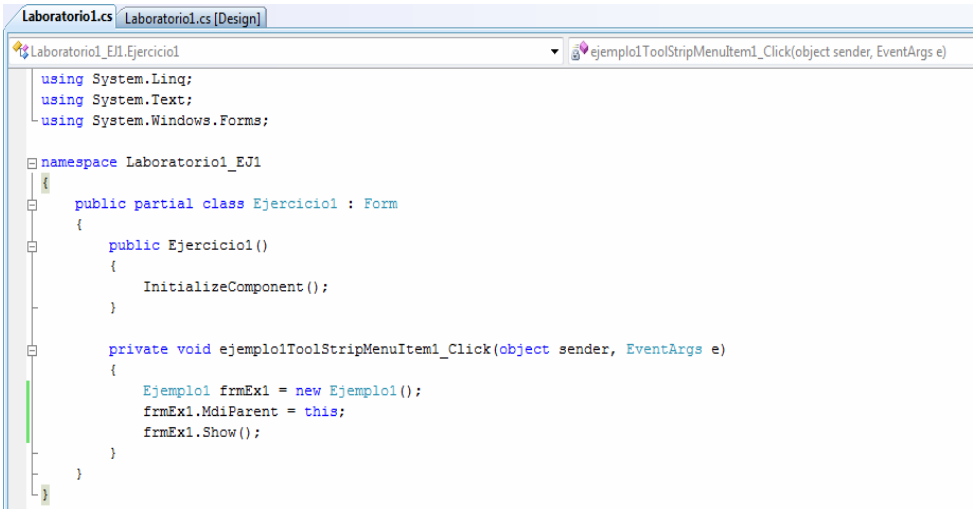

Figura 17: Programación del evento del *MenuStrip*.

10. Cuando ejecute por primera vez el programa podrá acceder al formulario hijo, haciendo *clic* en el menú correspondiente.

11. Abra nuevamente el formulario "Ejemplo1" y modifíquelo de forma tal que contenga un campo de texto, una lista y un botón, como se muestra en la figura 18.

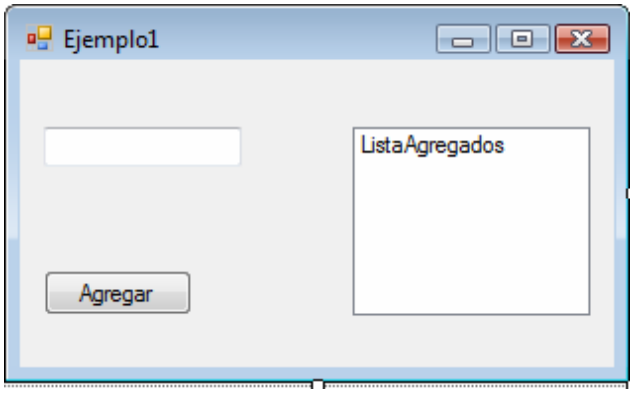

Figura 18: Formulario Ejemplo1.

12. Programe el evento *clic* del botón Agregar. Debe realizarlo de forma tal que el campo permita agregar solamente números y no cadenas de texto. Utilice el manejo de excepciones para controlar este requerimiento. A continuación, se muestra la solución al problema.

Solución

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Laboratorio1_EJ1
{
    public partial class Ejemplo1 : Form
    {
        public Ejemplo1()
        {
            InitializeComponent();
        }

        private void BotonAgregar_Click(object sender, EventArgs e)
        {
            int Valor = new int();
            try {
                Valor = System.Int32.Parse(ValorAgregar.Text);
                ListaAgregados.Items.Add(Valor);
                ValorAgregar.Text = "";
            }
            catch {
                MessageBox.Show("El valor que intenta agregar no es un
número. Intente de nuevo", "Ejemplo 1", MessageBoxButtons.OK);
            }
        }
    }
}
```

13. Agregue un nuevo botón al formulario del ejemplo 1 y programe el evento de forma tal que muestre una ventana con los valores de los elementos que están seleccionados en la lista. Intente hacerlo por usted mismo. A continuación, se muestra la solución.

Solución

```
private void MostrarSeleccionados_Click(object sender, EventArgs e){
   if (ListaAgregados.Items.Count != 0) {
      String items = "Los elementos seleccionados son: \n";
      foreach (Object Seleccionado in ListaAgregados.SelectedItems) {
         items += Seleccionado.ToString();
```

```
            items += "\n";
        }
        MessageBox.Show(items, "Ejemplo 1", MessageBoxButtons.OK);
    }
    else {
        MessageBox.Show("La lista está vacía. Debe agregar elementos
primero", "Ejemplo 1", MessageBoxButtons.OK);
    }
}
```

14. Seguidamente, resuma en un cuadro similar al cuadro número 7 las nuevas funciones, librerías y estructuras de control que ha conocido en el transcurso de este laboratorio.

Cuadro 7
Resumen de funciones y estructuras de control aprendidas en el laboratorio 1.1

Nombre	Tipo	Utilidad
foreach	Estructura de control	Recorre uno por uno los objetos de una colección.
MessageBox	Objeto	Muestra una ventana alternativa al usuario.
System.Int32.Parse	Función	Realiza un "*parse*" de un objeto determinado hacia un entero.
This	Objeto	Objeto que hace referencia a la clase que se encuentra actualmente en uso.

15. Guarde su solución en un disco externo, *flash usb* o disco compacto y llévelo a la tutoría #1. El laboratorio 1.2 se realizará con base en el laboratorio 1.1.

Laboratorio 1.2

Realice el laboratorio 1.2 utilizando como base el laboratorio 1.1.

1. Cree un nuevo objeto formulario y nómbrelo como "Ejemplo2".

2. Cree un nuevo elemento bajo la jerarquía del Menú principal que se llame Ejemplo 2.
3. Enlace el elemento del menú al formulario que acaba de crear, utilizando sus conocimientos en *MDI*. El formulario "Ejemplo2" debe ser una ventana hija del formulario "Ejercicio1".
4. Agregue los controles necesarios al formulario "Ejemplo2" de forma que cumpla con los siguientes requerimientos. Recuerde que debe usar los controles más adecuados para cada uno de los datos.
    a. Una cejilla para capturar los datos de un estudiante:
        i. Nombre y apellidos.
        ii. Cédula.
        iii. Sexo.
        iv. Fecha de nacimiento.
        v. Provincia donde vive.
    b. Una cejilla para capturar los datos académicos:
        i. Materia que cursa. Sólo puede seleccionar una, entre las siguientes:
            1. Historia.
            2. Filosofía.
            3. Cálculo.
            4. Literatura.
        ii. Verificación de estudiante activo o no.
        iii. Año que cursa.
    c. Cada una de las cejillas debe tener un botón de guardar datos. Controle que todos los datos estén llenos antes de guardarlos. El evento de guardar colocará dichos datos en un archivo de texto plano o *XML*.

Para aprender sobre el manejo de los archivos de texto en C#, así como del manejo de *XML*, consulte los capítulos 18 y 19 del libro de texto.

Usar los controles adecuados para capturar datos es una medida muy eficaz para reducir los errores de usuario. Limitar las opciones a las factibles es una buena práctica de programación.

Restringir en forma excesiva al usuario puede resultar frustrante para él, pero abrir sus posibilidades puede producir demasiados errores de entrada. Encuentre un balance a la hora de limitar las entradas por medio de controles. Dos problemas frecuentes en esta área son: dejar muchos campos de texto para ingresar datos y presentar demasiados controles en un solo formulario o cejilla. Nunca deje de lado la usabilidad en sus aplicaciones. Recuerde que la aplicación debe ser agradable y fácil de aprender.

5. Ahora se le presentan unas pantallas en las figuras 19 y 20 de ejemplo para la aplicación de este ejercicio.

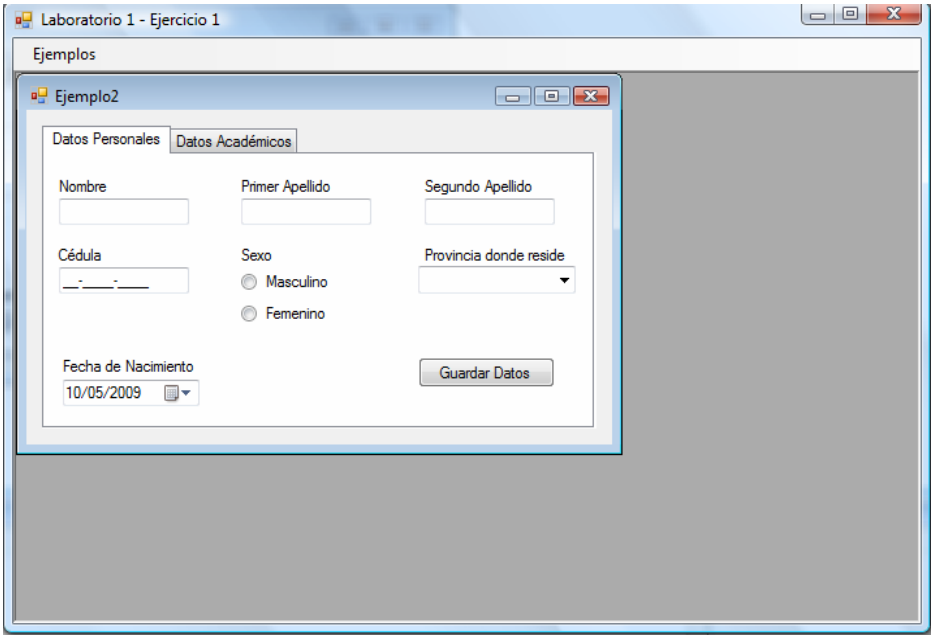

Figura 19: Cejilla inicial de la aplicación.

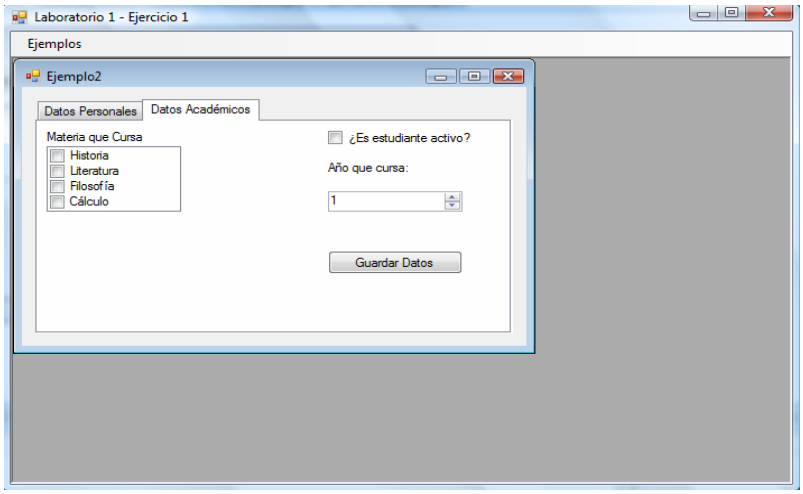

Figura 20: Cejilla secundaria de la aplicación.

6. Complete el cuadro número 8 de manera similar a como se realizó en el cuadro 7.

Cuadro 8
Resumen de funciones y estructuras de control aprendidas en el laboratorio 1.2

Nombre	Tipo	Utilidad

7. No olvide documentar sus aprendizajes adicionales en su reporte de laboratorio.

Respuestas a los ejercicios de autoevaluación

Ejercicios 1.1

El siguiente es un ejemplo de cómo puede programarse la clase Avión en C#.

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

namespace Avion
{
    public class Avion
    {
        private int potenciaMotores;
        private bool trenAbajo;
        private String alerones;
        private bool puertasAbiertas;
        private String estado;

        public Avion()
        {
            potenciaMotores = 0;
            trenAbajo = false;
            alerones = "Apagados";
            puertasAbiertas = false;
            estado = "En tierra";
        }
        private void despegar()
        {
            potenciaMotores = 100;
            trenAbajo = true;
            alerones = "Arriba";
            puertasAbiertas = false;
            estado = "Despegando";
        }

        private void volar()
        {
            potenciaMotores = 70;
            trenAbajo = false;
            alerones = "Horizontal";
            puertasAbiertas = false;
            estado = "En vuelo";
        }

        private void aterrizar()
        {
            potenciaMotores = -80;
            trenAbajo = true;
```

```
        alerones = "Abajo";
        puertasAbiertas = false;
        estado = "Aterrizando";
    }

    private void descargar()
    {
        potenciaMotores = 0;
        trenAbajo = true;
        alerones = "Apagados";
        puertasAbiertas = true;
        estado = "En tierra";
    }
    public int PotenciaMotores
    {
        get
        {
            return potenciaMotores;
        }
        set
        {
            potenciaMotores = value;
        }
    }
    public bool TrenAbajo
    {
        get
        {
            return trenAbajo;
        }
        set
        {
            trenAbajo = value;
        }
    }
    public bool PuertasAbiertas
    {
        get
        {
            return puertasAbiertas;
        }
        set
        {
            puertasAbiertas = value;
        }
    }
    public String Alerones
    {
        get
        {
            return alerones;
        }
        set
        {
            alerones = value;
        }
    }
```

```
        public String Estado
        {
            get
            {
                return estado;
            }
            set
            {
                estado = value;
            }
        }
    }
}
```

Ejercicios 1.2

Ejemplo de cómo quedaría la clase avión incluyendo la lógica de cambio de estado.

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

namespace Avion
{
    public class Avion
    {
        private int potenciaMotores;
        private bool trenAbajo;
        private String alerones;
        private bool puertasAbiertas;
        private String estado;

        public Avion()
        {
            potenciaMotores = 0;
            trenAbajo = false;
            alerones = "Apagados";
            puertasAbiertas = false;
            estado = "En tierra";
        }
        private void despegar()
        {
            potenciaMotores = 100;
            trenAbajo = true;
            alerones = "Arriba";
            puertasAbiertas = false;
            estado = "Despegando";
        }

        private void volar()
        {
```

```
        potenciaMotores = 70;
        trenAbajo = false;
        alerones = "Horizontal";
        puertasAbiertas = false;
        estado = "En vuelo";
    }

    private void aterrizar()
    {
        potenciaMotores = -80;
        trenAbajo = true;
        alerones = "Abajo";
        puertasAbiertas = false;
        estado = "Aterrizando";
    }

    private void descargar()
    {
        potenciaMotores = 0;
        trenAbajo = true;
        alerones = "Apagados";
        puertasAbiertas = true;
        estado = "En tierra";
    }
    public int PotenciaMotores
    {
        get
        {
            return potenciaMotores;
        }
        set
        {
            potenciaMotores = value;
        }
    }
    public bool TrenAbajo
    {
        get
        {
            return trenAbajo;
        }
        set
        {
            trenAbajo = value;
        }
    }
    public bool PuertasAbiertas
    {
        get
        {
            return puertasAbiertas;
        }
        set
        {
            puertasAbiertas = value;
        }
    }
```

```
        public String Alerones
        {
            get
            {
                return alerones;
            }
            set
            {
                alerones = value;
            }
        }
        public String Estado
        {
            get
            {
                return estado;
            }
            set
            {
                estado = value;
            }
        }
        public bool cambiarEstado(int estadoSolicitado) //Cambia el
estado de un Avión.
        {
            /*
             * Función: cambiarEstado
             * Utilidad: Cambia el estado actual del avión. Verifica que
el estado tenga un orden lógico y que no
             * se puedan hacer cambios abruptos en el estado del avión.
             * Parámetros de entrada: (integer) estadoSolicitado
             * Función de los parámetros de entrada:
             * - estadoSolicitado: Indica el estado en que se desea
colocar al avión
             *   - 0 : En tierra
             *   - 1 : Despegando
             *   - 2 : En vuelo
             *   - 3 : Aterrizando
             */
            bool cambio = false; //variable que indica si el cambio de
estado fue exitoso o no.
            switch (estadoSolicitado)
            {
                case 0:
                    if (estado == "Aterrizando")
                    {
                        descargar();
                        cambio = true;
                    }
                    break;
                case 1:
                    if (estado == "En tierra")
                    {
                        despegar();
                        cambio = true;
                    }
                    break;
```

```
                case 2:
                    if (estado == "Despegando")
                    {
                        volar();
                        cambio = true;
                    }
                    break;
                case 3:
                    if (estado == "En vuelo")
                    {
                        aterrizar();
                        cambio = true;
                    }
                    break;
            }
            return cambio;
        }
    }
}
```

Ejercicios 1.3

La siguiente es una propuesta visual de solución para el ejercicio 1.3. Usted puede trabajar sus interfaces gráficas a su gusto, siempre y cuando sean entendibles y usables. Por razones de espacio, la solución programada de esta pantalla se encuentra disponible en la plataforma virtual.

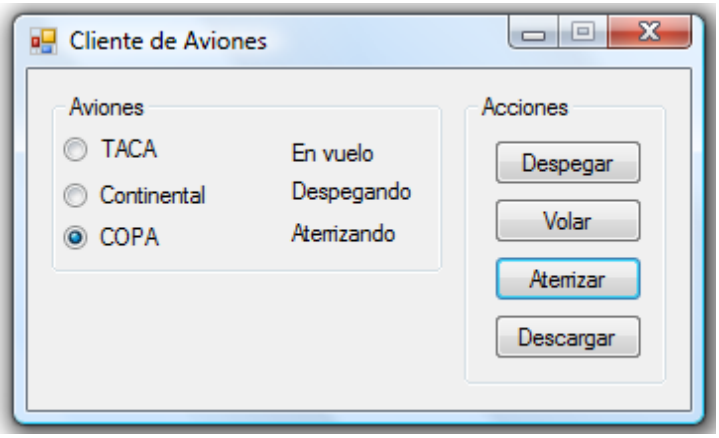

Figura 21: Ejemplo de pantalla de control de aviones.

Glosario

IDE. Siglas para *Integrated Development Environment*, entorno integrado de desarrollo. Aplicación de *software* que provee facilidades para su desarrollo.

GU. Siglas para *Graphic User Interface*, interfaz gráfica de usuario.

MDI. Siglas para *Multiple Document Interface*, interfaz de documentos múltiples.

SDI. Siglas para *Single Document Interface*, interfaz de documento simple.

III. Segunda sección (Segunda tutoría) Subprocesamiento múltiple

La segunda sección de esta guía contiene un único tema, que es uno de los enfoques principales del curso: el subprocesamiento múltiple. Con frecuencia se escucha que las computadoras son entes multiproceso, multi-hilo y multi-tareas. Pero, ¿qué significan estos conceptos? ¿Cómo se utilizan en las labores de un programador? Y, ¿cómo se plasman estos en el lenguaje de programación C#? La finalidad de esta parte es brindarle los recursos teóricos y prácticos para que pueda dar respuesta a esas preguntas.

Lea y estudie las secciones del tema 5 del libro, que se han destacado en la estructura de la guía que se encuentra en la página 8. Cuando haya leído las unidades del libro correspondientes, se le recomienda elaborar un cuadro comparativo a manera de resumen, con los principales conceptos aprendidos, y compararlos con los que se conocen de los lenguajes y cursos anteriores.

Tema 5: Conceptos importantes en el subprocesamiento múltiple

El procesamiento múltiple es la capacidad que tiene una computadora de ejecutar muchas tareas simultáneamente. Estas pueden ser programas completos o secciones de éstos. Cuando tenemos varias aplicaciones distintas, como *Word, Excel, Outlook* y *Firefox* abiertos al mismo tiempo en el escritorio, se dice que la computadora está ejecutando en modo multi-tarea. Este existe por defecto en los sistemas operativos modernos y difícilmente podríamos concebir que no fuera así. ¿Podría imaginarse un sistema operativo donde pueda ejecutar únicamente un sólo programa a la vez? Imposible. Esta situación se hace aún más visible en las aplicaciones *Web*. Un único programa servidor atiende a múltiples clientes simultáneos que lo acceden a través de Internet, brindando acceso por igual para

todos, sin que exista alguna especie de "cola de espera" que cause un retraso notable para el usuario. Un servidor *Web* utiliza el concepto de subprocesamiento múltiple*:* cuando un cliente lo contacta para realizar alguna actividad, éste crea y asigna un nuevo proceso hijo para que lo atienda. Cada uno de ellos está vigilado por el proceso padre o monitor, que controla los accesos concurrentes a los recursos compartidos, así como el ciclo de vida de cada uno de los procesos hijos.

Un subproceso puede crearse a partir de una sección específica del código o de un programa como un todo. Cada uno de ellos se ejecuta en lo que se conoce como un hilo de ejecución distinto. Los hilos no pueden ejecutarse por sí solos, ya que requieren un proceso padre o monitor para correr. Están asociados a uno en particular y también se conocen como procesos ligeros. Siempre corren dentro del contexto del proceso principal, compartiendo sus recursos de ejecución. La figura 22a muestra el concepto de multitarea, multihilos y subprocesamiento de una forma gráfica.

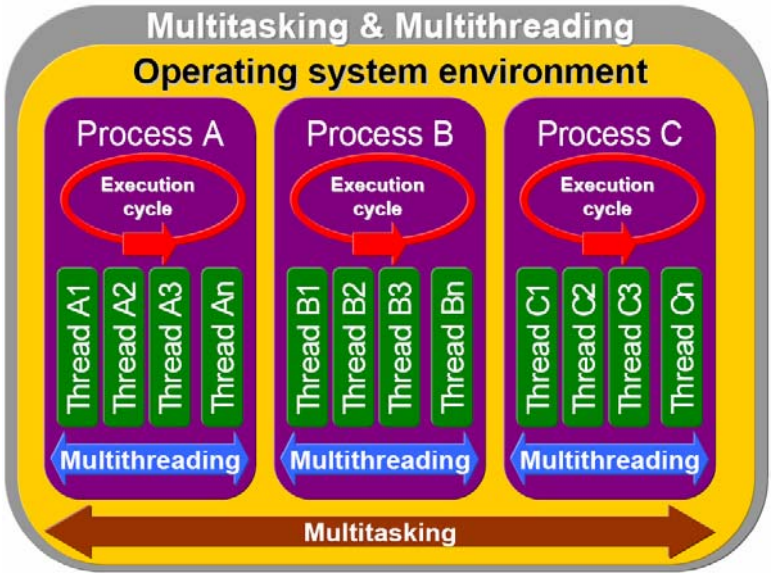

Figura 22a: Representación gráfica de los conceptos de multitarea, multihilos y subprocesamiento. Fuente: (Seda).

Para aclarar este concepto, piense en el programa de *Open Office Writer*. Este *software* presenta su pantalla de digitación, que sería su proceso principal. Adicionalmente, puede

tener un hilo en *background* chequeando automáticamente la gramática, otro la ortografía y otro más salvando automáticamente los cambios del documento en que usted se encuentra trabajando. Así como este *software*, cada aplicación puede correr varios hilos que realicen diferentes tareas.

Los subprocesos tienen dos atributos importantes: su prioridad y su estado. La prioridad de un proceso determina la "urgencia" y la importancia de éste. Gusto más prioridad tenga, mayor será el tiempo de procesador que se le asigne. Para entender mejor el concepto de partición de tiempo de procesador que poseen algunos sistemas operativos, haga énfasis en el segundo párrafo de la sección 15.3 del libro, página 513.

El estado de un subproceso determina la situación de ejecución actual del mismo. Su ciclo de vida es un conjunto de dos o más de estos estados. Estos indican si tiene algún tipo de requerimiento de recursos, o se encuentra suspendido o bloqueado. Observe el diagrama de estados de *UML* de la figura 15.1 del libro, página 512, para comprender el ciclo de vida de un subproceso. Se le presenta el cuadro 1 que contiene un resumen de los estados y prioridades de los procesos en C#.

Cuadro 1
Resumen de estado y prioridad de los subprocesos

Prioridad	**Estado**
Lowest	*Running*
BelowNormal	*Stopped*
Normal	*Aborted*
AboveNormal	*WaitSleepJoin*
Highest	*Suspended*
	Blocked

Práctica programada #2

Para practicar los conceptos aprendidos, haga el ejercicio de la figura 15.3 del libro, página 515.

1. Descargue el archivo *cap15.zip* del *CD-ROM* que se provee con el libro. Descomprima el archivo en su computadora y ubique el archivo *ProbadorSubprocesos.cs*.

2. Cree una nueva aplicación de consola en *Visual C# Express*.

3. Incluya el archivo *ProbadorSubprocesos.cs* en su proyecto.

4. Elimine el archivo *Program.cs* del proyecto.

5. Agregue la siguiente línea de código justo antes de la llave que finaliza el método *Main*.
    a. Console.ReadLine();

Esto le permitirá observar mejor los resultados de la ejecución del programa.

6. Las líneas de su programa deben verse así:

```
            Console.WriteLine( "Subprocesos iniciados\n" );
            Console.ReadLine();
        } // fin del método Main
    } // fin de la clase ProbadorSubprocesos
```

7. Ejecute el programa.

8. Documente las principales funciones y su utilidad en una tabla resumen como la de la página 17 de esta guía de estudio y adjúntela a su tarea corta #2.

Sincronización de subprocesos

Cuando un proceso padre o servidor posee múltiples subprocesos, es común que estos necesiten recursos compartidos de manera simultánea. Este acceso concurrente debe ser controlado en pro de evitar lo que se conoce como un *"deadlock"* o bloqueo mutuo. Un *deadlock o interbloqueo,* es el bloqueo permanente de los subprocesos de un sistema que maneja accesos concurrentes a recursos compartidos.

Los subprocesos se disputan estos recursos de forma simultánea, de manera que ninguno puede continuar su ejecución normal. Cuando esto sucede, no hay una solución posible y, por lo tanto, debe cerrarse el proceso principal sin que los subprocesos hayan finalizado. Un ejemplo de la vida real está dado en la siguiente cita:

"Todos los interbloqueos surgen de necesidades que no pueden ser satisfechas, por parte de dos o más procesos. En la vida real, un ejemplo puede ser el de dos niños que intentan jugar al arco y flecha, uno toma el arco, el otro la flecha. Ninguno puede jugar hasta que alguno libere lo que tomó." (Meza, 2005)

La figura 22b representa un bloqueo mutuo por medio de grafos. La figura 22c lo representa por medio de utilización de recursos.

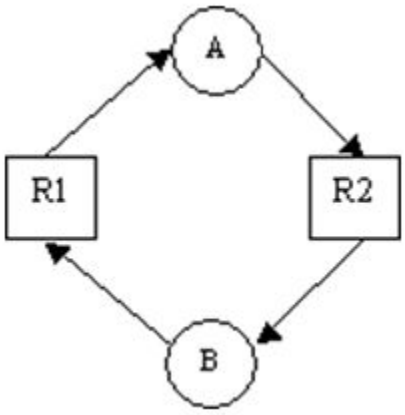

Figura 22b: Un bloqueo mutuo representado por grafos.
Fuente: <http://es.wikipedia.org/wiki/Bloqueo_mutuo>.

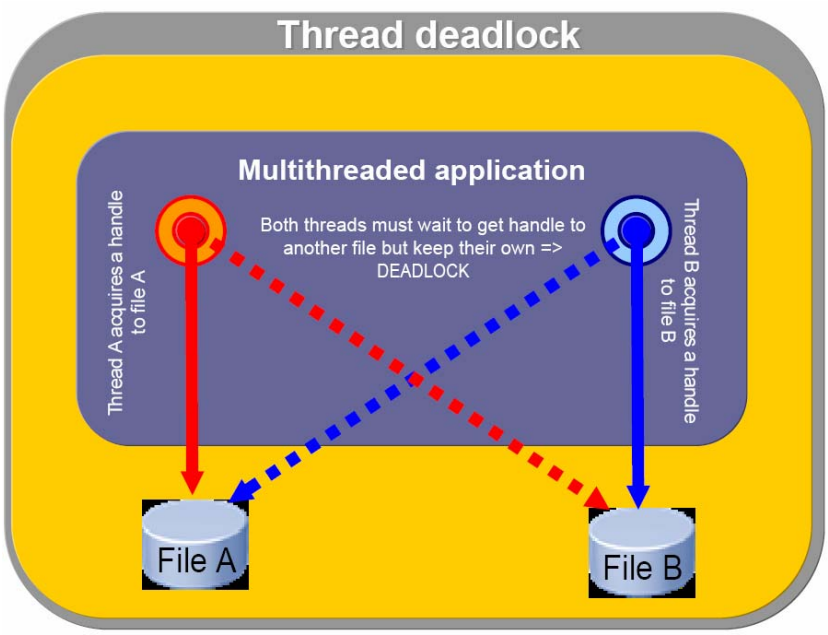

Figura 22c: Un bloqueo mutuo representado por un gráfico de recursos.
Fuente: (Seda).

En la figura 22b, observe cómo el proceso B tiene asignado el recurso *R2*, pero necesita del recurso *R1*, que está asignado al proceso A**,** que a su vez está solicitando el recurso *R2*. Como puede notarlo, no hay una salida posible en una situación como esta.

Para evitar que sucedan los bloqueos mutuos, los lenguajes de programación modernos implementan la sincronización de procesos. La instrucción *lock* de C# le permite realizar un bloqueo de recursos compartidos para los subprocesos. Las secciones donde se utiliza la instrucción *lock* deben delimitar aquellas donde se acceden los recursos que todos los subprocesos necesitan, tales como archivos, bases de datos, pantallas, variables o memoria.

Cuando se trabaja con interfaces de usuario, la sincronización de procesos tiene la misma importancia que con programas desde la consola. En la sección 15.9 del libro de texto, páginas 539 a 543 se explica este concepto en interfaces gráficas. Note cómo es importante no solamente el bloqueo de los recursos de acceso concurrente sino también en los elementos de la interfaz visual.

Lea con detenimiento los errores comunes de programación, *tip* de prevención de errores y *tip* de rendimiento que se encuentran en las páginas 518 y 519 del libro de texto. Le ayudarán a complementar de una mejor forma los conceptos relacionados con la sincronización de procesos.

En esta sección de la guía únicamente se trabajará la sincronización de procesos, y no las relaciones de productor/consumidor. Para ampliar sus conocimientos en esta otra forma de sincronización, lea las secciones 15.6 a 15.8 del libro de texto, páginas 519 a 539.

Sincronización con semáforos

Este apartado ha sido incluido de manera adicional, ya que el libro de texto no trata este tema. Un semáforo "es un objeto que limita el número de subprocesos que pueden tener acceso a un recurso o grupo de recursos simultáneamente" (Microsoft, Inc.). Básicamente, mediante un semáforo podemos definir cuáles objetos son recursos compartidos, sin necesidad de utilizar las secciones *lock* de C#. La siguiente cita textual del sitio de *MSDN* explica con detenimiento el funcionamiento de ellos en C#.

"Utilice la clase ***Semaphore*** para controlar el acceso a un grupo de recursos. Para entrar en el semáforo, los subprocesos llaman al método ***WaitOne***, que se hereda de la clase ***WaitHandle***, y para liberarlo llaman al método ***Release***".
El recuento de un semáforo disminuye cada vez que un subproceso entra en él, e incrementa cuando un subproceso lo libera. Cuando el recuento es cero, las solicitudes posteriores se bloquean hasta que otros subprocesos liberan el semáforo. Cuando todos los subprocesos han liberado el semáforo, el recuento está en el valor máximo especificado al crear el semáforo.
No hay ningún orden garantizado, como *FIFO o LIFO*, para la entrada de los subprocesos bloqueados en el semáforo.
Un subproceso puede entrar varias veces en el semáforo, mediante llamadas al método ***WaitOne***. Para liberar algunas de estas entradas o todas ellas, el subproceso puede llamar varias veces a la sobrecarga del método ***Release*** sin parámetros, o bien a la sobrecarga del método ***Release(Int32)***, que especifica el número de entradas que se liberan.

La clase ***Semaphore*** no exige la identidad del subproceso en las llamadas a ***WaitOne*** o ***Release***. Es responsabilidad del programador garantizar que los subprocesos no liberen el semáforo demasiadas veces. Por ejemplo, supongamos que un semáforo tiene un recuento máximo de dos y que entran en el semáforo los subprocesos A y B. Si un error de programación del subproceso B hace que llame a ***Release*** dos veces, las dos llamadas tienen éxito. Se alcanza el recuento máximo del semáforo y, si el subproceso A llama posteriormente a ***Release***, se genera una excepción ***SemaphoreFullException.***" (Microsoft, Inc.)

Observe el ejemplo 1 de utilización de semáforos, tomado de la biblioteca de ejemplos de Microsoft *MSDN*. Si lo desea, puede copiarlo y ejecutarlo en un nuevo proyecto de aplicación de consola.

Ejemplo 1

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

namespace Semaforos
{
    public class Example
    {
        // A semaphore that simulates a limited resource pool.
        //
        private static Semaphore _pool;

        // A padding interval to make the output more orderly.
        private static int _padding;

        public static void Main()
        {
            // Create a semaphore that can satisfy up to three
            // concurrent requests. Use an initial count of zero,
            // so that the entire semaphore count is initially
            // owned by the main program thread.
            //
            _pool = new Semaphore(0, 3);

            // Create and start five numbered threads.
            //
            for (int i = 1; i <= 5; i++)
            {
                Thread t = new Thread(new
ParameterizedThreadStart(Worker));

                // Start the thread, passing the number.
                //
```

```
                t.Start(i);
            }

            // Wait for half a second, to allow all the
            // threads to start and to block on the semaphore.
            //
            Thread.Sleep(500);

            // The main thread starts out holding the entire
            // semaphore count. Calling Release(3) brings the
            // semaphore count back to its maximum value, and
            // allows the waiting threads to enter the semaphore,
            // up to three at a time.
            //
            Console.WriteLine("Main thread calls Release(3).");
            _pool.Release(3);

            Console.WriteLine("Main thread exits.");
        }
        private static void Worker(object num)
        {
            // Each worker thread begins by requesting the
            // semaphore.
            Console.WriteLine("Thread {0} begins " +
                "and waits for the semaphore.", num);
            _pool.WaitOne();

            // A padding interval to make the output more orderly.
            int padding = Interlocked.Add(ref _padding, 100);

            Console.WriteLine("Thread {0} enters the semaphore.", num);

            // The thread's "work" consists of sleeping for
            // about a second. Each thread "works" a little
            // longer, just to make the output more orderly.
            //
            Thread.Sleep(1000 + padding);

            Console.WriteLine("Thread {0} releases the semaphore.", num);
            Console.WriteLine("Thread {0} previous semaphore count: {1}",
                num, _pool.Release());
        }
    }
}
```

El ejemplo 1 crea un semáforo con 3 “espacios”. El nombre representa el objeto por el cual se está en espera. Luego inicia subprocesos que se bloquean en espera de la señal del semáforo. Cuando un objeto llama a la función *WaitOne,* el proceso detiene su ejecución hasta que otro subproceso realice la función *Release* sobre el mismo semáforo. Esto sería el equivalente a que varios procesos lleguen de forma simultánea a la instrucción/sección *lock*

de un programa e inician una cola de espera. La señal de un semáforo indica que el siguiente proceso puede liberarse y seguir trabajando o morir.

En el ejemplo 1, el proceso principal coloca el valor del semáforo en su mayor capacidad utilizando el método *Release(Int 32),* donde *Int*32 representa al número de procesos que pueden esperar en él. Los procesos *Worker* realizan un "trabajo" que consiste en dormir por aproximadamente un segundo y llaman a la función *Release( )* para liberarlo. Esto daría espacio a procesos adicionales a entrar en la cola de espera de él. Si en algún momento la cuenta de campos del semáforo alcanza cero, éste se bloquea hasta que otro subproceso libere una entrada en el mismo.

En el cuadro 2 se resumen las principales funciones de los semáforos en C#:

Cuadro 2
Funciones de los semáforos

Método	**Utilidad**
Semaphore sem1 = *new Semaphore(valorInicio, valorMaximo)*	Constructor de la clase.
Sem1.WaitOne()	El proceso entra al semáforo y decrementa los campos de espera.
Sem1.Release()	Libera un campo de espera del semáforo.
Sem1.Release(Int32 n)	Libera "*n*" campos de espera del semáforo.

Ejercicios de autoevaluación 2.1

Conteste las siguientes preguntas:

1. ¿Cuál es la biblioteca utilizada para la sincronización de los subprocesos en C#?

   _____

2. Defina con sus propias palabras el concepto de bloqueo mutuo y escriba un ejemplo donde dos subprocesos podrían tener esta situación.

   _____
   _____
   _____
   _____
   _____
   _____
   _____

3. Explique la funcionalidad de la clase Monitor y 2 de sus funciones.

   _____
   _____
   _____
   _____
   _____
   _____
   _____

4. Explique ¿Qué función tiene un semáforo en la sincronización de procesos y en qué consisten las funciones *WaitOne y Release*?

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

5. Cree un nuevo proyecto en *Visual* C# y cargue en él los ejemplos de las figuras 15.13 y 15.14 del libro, páginas 540 y 542. Ejecute la aplicación y observe los resultados. Dada la complejidad del ejercicio, la solución que incluye el proyecto completo junto con su ejecución la encontrará en la plataforma virtual.

Laboratorios

Laboratorio 2.1

En el laboratorio 2.1 se estudia el concepto de sincronización de procesos mediante semáforos. Para ello se resolverá uno de los problemas clásicos de este tema. El problema del *"barbero dormilón"*. Lea con detenimiento el enunciado.

"En una peluquería hay un único barbero, una única silla para cortar el pelo y *n* sillas para que los clientes esperen su turno. Usted puede definir el número de sillas de espera a su gusto. Cuando no hay clientes en la peluquería, el barbero se duerme en la silla de cortar el pelo. Cuando llega un cliente, debe despertar al barbero y sentarse en la silla para que le corten el pelo. Si llegan más clientes mientras uno está siendo atendido, deben esperar en las sillas dispuestas para ello. Si no hay más sillas para esperar, el cliente se va."

Programe al barbero y a los clientes de forma tal que no se bloqueen y que no entren en condición de competencia.

 No se dedique a programar si aún no ha pensado en la forma en que resolverá una situación. Hacerlo "en caliente" es una mala práctica que cierra el pensamiento a nuevas soluciones. Trabaje primero en la lógica del programa, realice mini especificaciones, pseudocódigos y recapacite además en la optimización de su solución. Cuando haya superado estos pasos, programe el código necesario.

1. Cree un nuevo proyecto de *Visual C# Express* y cree una aplicación de consola.

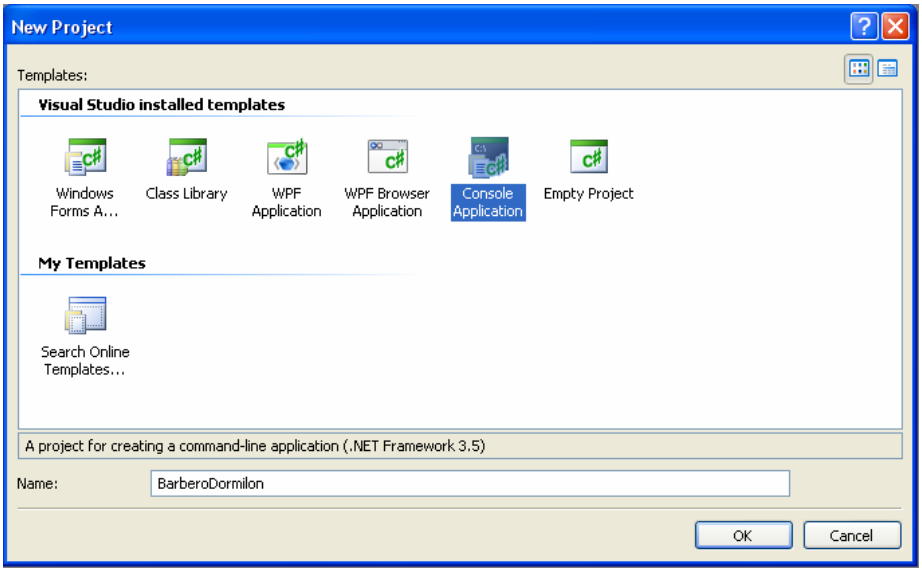

Figura 23: Creación de un nuevo proyecto de consola.

2. En el explorador de soluciones, renombre el archivo *Program.cs* como Barbero.cs.

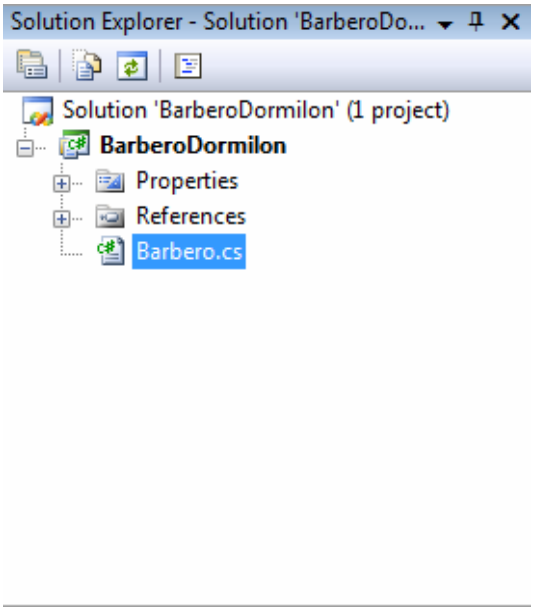

Figura 24: Explorador de soluciones.

3. Como atributos de la clase, cree los semáforos que representen los recursos compartidos de la barbería, así como todas las variables adicionales que considere necesarias. A continuación, se adjunta el ejemplo 2. Adicionalmente, cree dos constantes que representen al número máximo de clientes y el número de sillas de espera.

Ejemplo 2

```
// Número aleatorio que indica cuánto se tarda cortando el pelo a un
cliente
        static Random Rand = new Random();
        // Definen el número máximo de clientes por día y sillas de
espera
        const int maxClientes = 25;
        const int numSillas = 5;
        static Semaphore cuartoEspera = new Semaphore(numSillas,
numSillas);
        static Semaphore sillaBarbero = new Semaphore(1, 1);
        static Semaphore almohadaBarbero = new Semaphore(0, 1);
        static Semaphore cinturonSilla = new Semaphore(0, 1);
        // Indicador de que todos los clientes ya están listos
        static bool todosListos = false;
```

4. Cree una clase Barbería que contenga tres funciones que representen al barbero, a los clientes y la función *Main*.

5. Programe la lógica del barbero. El siguiente bloque de código lo realiza de la siguiente manera:
    a. Mientras no estén todos los clientes listos, la primera acción que realiza el barbero es dormir. Para representarla se suma un proceso al semáforo de la almohada.
    b. Cuando corta el pelo, tarda un tiempo aleatorio y luego suelta el cinturón de la silla de barbero.
    c. Finalmente, si ya todos los clientes han sido atendidos, el barbero se va a su casa.

Ejemplo 3: Lógica del barbero

```
static void Barber()
{
    while (!todosListos)
    {
        Console.WriteLine("El barbero esta durmiendo...");
        almohadaBarbero.WaitOne();
        if (!todosListos)
        {
    // Cortando el pelo al cliente por un tiempo aleatorio de tiempo
            Console.WriteLine("El barbero está cortando el
cabello...");
            Thread.Sleep(Rand.Next(1, 3) * 1000);
            Console.WriteLine("El barbero terminó de cortar el
cabello...");
            cinturonSilla.Release();
        }
        else
        {
            Console.WriteLine("El barbero se va a casa por ahora...");
        }
    }
    return;
}
```

6. Programe la lógica del cliente. El siguiente bloque de código lo realiza de la siguiente manera:
    a. Cuando el cliente se crea, se asume que va saliendo de su casa.
    b. El hilo se "duerme" mientras el cliente hace su viaje hasta la barbería.
    c. Cuando el cliente llega a la sala, se coloca en la cola del cuarto de espera. Adicionalmente se coloca en la cola de la silla de barbero.
    d. Cuando se ha terminado la espera en la silla de barbero, (cuando otro cliente la desocupa) entonces libera también el recurso del cuarto de espera.
    e. Si el barbero está durmiendo, el cliente lo despierta, liberando el recurso de la almohada.
    f. Cuando el barbero se despierta, se coloca el cinturón de la silla, que sólo es liberado por el barbero cuando termina de cortar el pelo.
    g. Cuando el barbero libera el semáforo del cinturón de la silla, el cliente libera la silla del barbero.

Ejemplo 4: Lógica del cliente

```
static void Customer(Object number)
{
    int Number = (int)number;
    Console.WriteLine("El cliente {0} sale de su casa hacia la
barbería.", Number);
    Thread.Sleep(Rand.Next(1, 5) * 1000);
    Console.WriteLine("El cliente {0} ha llegado.", Number);
    cuartoEspera.WaitOne();
    Console.WriteLine("El cliente {0} está entrando a la sala de
espera", Number);
    sillaBarbero.WaitOne();
    cuartoEspera.Release();
    Console.WriteLine("Barbero, el cliente {0} desea cortarse el
cabello. ¡Levántese!", Number);
    almohadaBarbero.Release();
    cinturonSilla.WaitOne();
    sillaBarbero.Release();
    Console.WriteLine("El cliente {0} deja la barbería.", Number);
}
```

7. Programe la función *Main*. Esta debe crear la cantidad de hilos igual al número máximo de clientes por atender y finalizar el proceso del barbero.

    a.

```
static void Main()
{
    Thread BarberThread = new Thread(Barber);
    BarberThread.Start();
    Thread[] Customers = new Thread[maxClientes];
    for (int i = 0; i < maxClientes; i++)
    {
         Customers[i] = new Thread(new
      ParameterizedThreadStart(Customer));
        Customers[i].Start(i);
    }
    for (int i = 0; i < maxClientes; i++)
    {
        Customers[i].Join();
    }
    todosListos = true;
    almohadaBarbero.Release();
    // Esperando a que el proceso del barbero termine para finalizar
el programa
    BarberThread.Join();
    Console.WriteLine("Fin del programa. Esta debería ser la última
línea desplegada");
}
```

8. Construya la solución y ejecute el programa, como en las figuras 25 y 26.

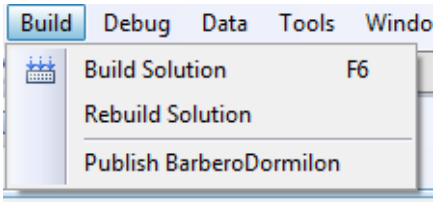

Figura 25: Construcción de una solución.

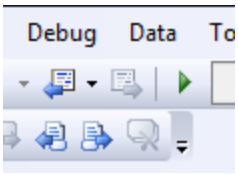

Figura 26: Ejecutar una aplicación.

9. Recuerde que puede descargar la solución programada desde la plataforma virtual en caso de que tenga alguna duda.

Laboratorio 2.2

1. Programe una solución para el problema de la figura 27. Puede resolverlo utilizando bloques de exclusión (*lock*) o con semáforos. Se presenta ahora el enunciado de otro de los problemas de sincronización de procesos más famosos: La cena de los filósofos.

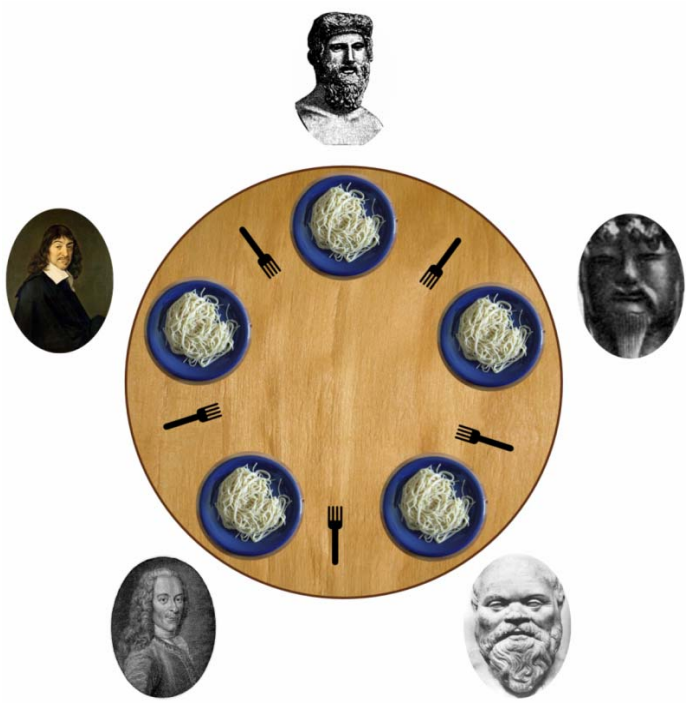

Figura 27: la cena de los filósofos.

"Cinco filósofos se sientan alrededor de una mesa y pasan su vida cenando y pensando. Cada filósofo tiene un plato de fideos y un tenedor a la izquierda de su plato. Para comer los fideos, son necesarios dos tenedores, y cada filósofo sólo puede tomar los que están a su izquierda y derecha. Si cualquier filósofo coge un tenedor y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda coger el otro tenedor, para luego empezar a comer. Si dos filósofos adyacentes intentan tomar el mismo tenedor a una vez, se produce una condición de carrera: ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer. Si todos los filósofos cogen el tenedor que está a su derecha al mismo tiempo, entonces todos se quedarán esperando eternamente, porque alguien debe liberar el tenedor que les falta. Nadie lo hará porque todos se encuentran en la misma situación (esperando que alguno deje sus tenedores). Entonces los filósofos se morirán de hambre, por lo que se produce un *deadlock*." (Tanenbaum, 2001)

2. El ejercicio #2 consiste en encontrar un algoritmo que permita que los filósofos nunca se mueran de hambre.

3. Programe una solución similar al ejercicio #1, en una aplicación de consola.

4. No olvide documentar sus aprendizajes en sus reportes de laboratorio.

5. Una solución a este problema se encuentra en la plataforma virtual. Puede descargarla para compararla con la suya o como ayuda adicional.

Glosario

MSDN. Siglas para *Microsoft Development Network*, red de desarrollo de Microsoft. Sitio en línea que contiene las ayudas a toda la plataforma de desarrollo de Microsoft *.NET*.

FIFO. Siglas para *"First In, First Out"*, primero en entrar primero en salir. Algoritmo de manejo de colas donde un objeto que ingresa de primero a la cola, es el primero en ser atendido.

LIFO. Siglas para *"Last In, First Out"*, último en entrar primero en salir. Algoritmo de manejo de pilas donde un objeto que ingresa de último a la pila, es el primero que sale de ella.

IV. Tercera sección (Tercera tutoría) Bases de datos y colecciones

La tercera sección de la guía se enfoca en los temas de bases de datos y colecciones. Ambos son comunes en el ámbito del desarrollo de aplicaciones y con el cual el profesional en informática tiene un contacto constante. Los lenguajes de programación modernos han evolucionado junto con los motores de bases de datos para que el enlace, extracción y manipulación con estos sea cada vez más sencillo de realizar, sin dejar de ser muy poderoso. C# no escapa a esta característica y permite establecer comunicación desde los programas desarrollados hacia casi cualquier tipo de base de datos de una forma transparente y con muchas ayudas que facilitan la labor.

Las colecciones, por su parte, son relativamente nuevas y se han convertido en pilar de los lenguajes de programación para el manejo de bloques o conjuntos de información en memoria de manera ágil y eficiente.

Cabe destacar que para el tema de bases de datos, el curso y la presente guía se enfocan específicamente en la utilización e implementación de los objetos de *ADO .NET* para conexión con ellas en las aplicaciones de C#. Se asume que el estudiante posee el conocimiento suficiente en normalización y manipulación de bases de datos, así como en su diseño, conceptos, características, principales funciones, procedimientos almacenados, *triggers* y transacciones, entre otros.

Tema 6: Bases de datos y componentes *ADO .NET*

Antes de adentrarse en la manipulación de bases de datos desde C#, lea las secciones 20.1 y 20.2 del libro páginas 726 y 727. Son un refrescamiento de lo estudiado en otros cursos

acerca de ellas y le serán de gran ayuda para lo que resta de este tema. Se repasan ahora algunos conceptos importantes de estas secciones del libro.

Si desea ampliar aún más sus conocimientos acerca de la tecnología *ADO .NET* en el lenguaje C#, puede consultar las páginas 230 a 243 del libro "**C#:** *Your visual blueprint for building .NET Applications"* de Eric Buttow y Tommy Ryan, que se encuentra disponible en la plataforma virtual.

Bases de datos relacionales

La figura 28 representa un diagrama relacional de la base de datos TALLER. Observe los nombres de sus tablas, sus atributos, llaves primarias y llaves foráneas.

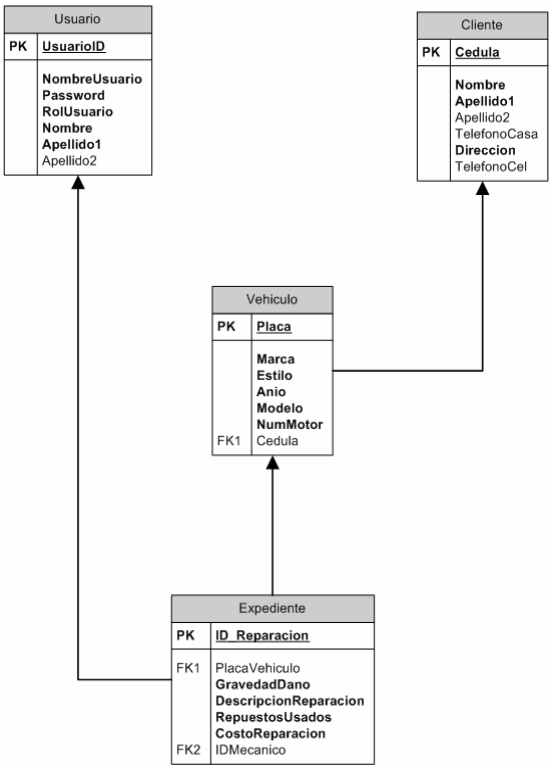

Figura 28: Diagrama de base de datos relacional.

Entre los objetos que se encuentran en el diagrama se observan:

- Tablas: Expediente, Usuario, Cliente, Vehículo.
- Llaves primarias: ID_Reparación, Usuario ID, Placa, Cédula.
- Llaves externas: PlacaVehiculo_FK_Placa, Vehiculo.Cedula_FK_Cedula, entre otras.

En caso de que no recuerde con exactitud las cláusulas y consultas principales de *SQL*, lea la sección 20.4 del libro de texto, páginas 731 a 739. En ellas se explican las cláusulas y consultas *INSERT*, *SELECT*, *DELETE*, *UPDATE*, *ORDER BY* e *INNER JOIN*.

Objetos *ADO .NET* para el acceso a bases de datos

ADO .NET es una *API* de acceso a datos. Provee diversos objetos para realizar conexiones y manipulación hacia diversos orígenes de ellos, entre los que se destacan: aplicaciones, bases de datos relacionales y archivos *XML*.

Su espacio de nombres principal es *System.Data*, y de aquí se desprenden dos más de nombre hijos. *System.Data.SqlClient*, que proporciona funciones optimizadas para Microsoft *SQL Server* y *System.Data.OleDb*, que brinda otras para acceder a cualquier fuente de datos.

Estos espacios de nombres proporcionan objetos para la manipulación de datos de manera desconectada y conectada, donde puede ser necesaria una conexión persistente a la base de datos o no.

En el cuadro 1 se resumen los principales objetos y características de *ADO .NET*.

Cuadro 1
Objetos de *ADO .NET*

Objeto *ADO .NET*	**Utilidad**
DataTable	Representa una tabla de datos. Contiene una colección de objetos *DataRow* y una colección de objetos *DataColumn*, que representan las filas y columnas de una tabla en una base de datos.
DataSet	Representa una estructura de tablas de una base de datos, junto con sus relaciones. Es una imagen en caché de los datos de una base de datos. Contiene una colección de *DataTables*, representando cada una de las tablas. Se conecta al origen de datos para poblar el objeto y cuando debe hacer actualizaciones.
SqlDataAdapter	Objeto que contiene una conexión a un origen de datos. Este objeto abre y cierra dicha conexión cuando es necesario actualizar o leerlos del origen. La actualización de ellos se realiza utilizando objetos *SqlCommand*.
DataReader	Utilizado para datos de sólo lectura y sólo hacia adelante. Mantiene siempre una conexión activa con el origen de ellos. Se puebla de datos al llamar al método *executeReader* del objeto *SqlCommand*.
DataGridView	Muestra los datos organizados en filas y columnas correspondientes a las filas y columnas del origen de datos al que se encuentra enlazado.
SqlCommand	Objeto que representa un comando de *SQL*. Puede contener parámetros y definir qué tipo de instrucción o comando representa (*store procedure, table direct, text).*

Antes de manipular una base de datos con objetos *ADO.NET*, tenga a mano su diagrama relacional. No se aventure apresuradamente a manipularla cuando no conoce los detalles del origen con el que está trabajando. Distinga las tablas, restricciones, relaciones, procedimientos almacenados y funciones con la que esta cuenta. Este conocimiento previo hará que pueda aprovechar mejor los objetos y funciones para cuestiones de rendimiento y sencillez de su aplicación.

Aunque el curso manipula únicamente bases de datos relacionales que utilizan el motor *SQL Server 2008*, recuerde que *ADO.NET* brinda facilidades para conectarse a cualquier origen de datos, tales como *Oracle* o *Sybase*. Utilice el espacio de nombres *System.Data.OleDB* para trabajar con estos orígenes. Los objetos dentro de ese espacio de nombres son casi idénticos a los que se encuentran dentro de *System.Data.SqlClient*.

Práctica programada #3

Para practicar los conceptos aprendidos, ejecute los ejercicios de las secciones 20.6, 20.7, 20.8 y 20.9 del libro de texto, páginas 740 a 759. Adjunte esta práctica a su tarea corta #2.

1. Obtenga el archivo *cap20.zip* del *CD-ROM* que se provee con el libro. Descomprima el contenido en una carpeta en su disco duro.
2. Cree un proyecto en *Visual* C# y guárdelo en sus proyectos como *EjerciciosCap20ADO*.
3. Cree una aplicación *MDI* en la que realice todos los ejercicios de las secciones 20.6, 20.7, 20.8 y 20.9 del libro, páginas 740 a 759.
4. En la plataforma *Web* podrá encontrar un Proyecto de *Visual* C# con todos los ejercicios resueltos.

Si desea obtener más ejemplos de utilización de *ADO .NET*, descargue de la plataforma virtual el archivo *CSharp.msi* que contiene los "*101 C# Samples*" de *MSDN*. Instale los ejemplos y refiérase al directorio de *"Data Access"*. Puede reforzar los conceptos de *ADO .NET* siguiendo las lecciones del *"The C# Station ADO.NET Tutorial"* que se encuentra en la siguiente dirección en Internet: <http://www.csharp-station.com/Tutorials/AdoDotNet/Lesson01.aspx>.

Ejercicios de autoevaluación 3.1

Conteste las siguientes preguntas

1. ¿Por qué el modelo de manipulación de datos de *ADO .NET* se conoce como modelo desconectado?

_____

_____

_____

_____

_____

2. Mencione 2 ventajas que presenta el modelo sin conexión de *ADO.NET*.

_____

_____

_____

_____

_____

_____

3. Escriba a continuación un ejemplo de código para ejecutar un comando de tipo *store procedure* que retorna un valor en una base de datos llamada "Universidad".

_____

_____

_____

_____

4. ¿Qué es una base de datos relacional y por qué son sencillas de usar con *ADO.NET*?

_____

_____

_____

_____

_____

_____

Tema 8: Colecciones

Se ha decidido cambiar el orden de los temas debido a que las colecciones tienen relación con la manipulación de datos y por lo tanto resulta más transparente su enseñanza luego de aprender sobre *ADO.NET* en C#.

Hasta hace algunos años, el manejo de varios objetos de una misma clase que estaban relacionados entre sí, se realizaba a través de colecciones "personalizadas", que no eran más que listas y pilas de punteros, así como vectores de objetos. El programador estaba entonces obligado a implementar todas las funciones de ellas utilizando conceptos avanzados como la sobrecarga de operadores y los iteradores. Esto implicaba sin duda alguna un trabajo extra. Si los objetos eran todos de distintos tipos no polimórficos, debía programar las instrucciones para cada una de las colecciones.

Las colecciones modernas, introducidas en el *Framework* de Microsoft *.NET*, facilitan el accionar de los programadores que deben manejar conjuntos de datos especializados y permiten crear listas, pilas y vectores de objetos sin necesidad de implementar todas las funcionalidades y operadores de la colección, disminuyendo así los tiempos de desarrollo y facilitando el manejo dentro de los programas.

Para este tema, es necesario que lea las secciones 26.1, 26.2, 26.4 y 26.5 del libro de texto, páginas 1018 a 1041. Como en las anteriores secciones de esta guía, se le recomienda elaborar un cuadro comparativo a manera de resumen, con los principales conceptos aprendidos, y compararlos con los que se conocen de los lenguajes y cursos anteriores. Si tiene alguna duda en particular, recuerde que existen los foros en la plataforma virtual, donde puede plantear sus consultas, que serán respondidas por los tutores encargados o por algún compañero. Esta es siempre la forma más eficiente de compartir conocimientos.
Estudie con detenimiento la figura 26.1 del libro de texto páginas 1019. Las interfaces que ahí se detallan son las clases "padre" de las colecciones que se explican más adelante. Observe que estas interfaces brindan los métodos básicos, y su especialización se realiza en las clases que heredan/derivan de ellas.

Para aclarar el concepto de una colección, piense en la clase Avión con la que se ha trabajado en el desarrollo de esta guía y que son los controladores aéreos de un aeropuerto y que tienen un programa que les permite saber cuáles aviones están en vuelo actualmente.

Cuando un avión se acerca, su radar emite una señal con el número de placa del Avión. Esta es detectada por el programa y de inmediato coloca un nuevo objeto en pantalla para que el controlador aéreo lo visualice.

Para facilitar el manejo de los objetos de tipo Avión en memoria y su manipulación dentro del programa, puede utilizar dos tipos de colecciones: Una genérica tipo lista o una no genérica tipo tabla *hash*. Una lista permite enlazar todos los objetos de tipo Avión uno detrás de otro. Una tabla *hash* manipula los objetos utilizando como llave *hash* el número de placa del avión.

La figura 29 muestra un ejemplo gráfico de cómo se puede "visualizar" una lista en memoria.

Figura 29: Representación de una lista de objetos Avión.

Recuerde que cada uno de los cuadros corresponde a un Avión independiente, donde cada uno de ellos tiene todos los atributos y funciones de la clase. Esto brinda la facilidad de recorrer la lista para todos los objetos con la estructura de control *foreach*, como se muestra en el ejemplo 1.

Ejemplo 1

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

namespace ListaAviones
{
    class ListaAviones
    {
        static void Main(string[] args)
        {
            Avion.Avion avionTaca = new Avion.Avion();
            Avion.Avion avionContinental = new Avion.Avion();
            Avion.Avion avionCopa = new Avion.Avion();

            List <Avion.Avion> listaAviones = new List<Avion.Avion>();
            listaAviones.Add(avionTaca);
            listaAviones.Add(avionContinental);
            listaAviones.Add(avionCopa);

            foreach (Avion.Avion avionItem in listaAviones) {
                Console.Write("Estado de los aviones: {0}\n",
avionItem.Estado);
                Console.ReadLine();
            }
        }
    }
}
```

Observe en la figura 30 como el *Intellisense* de *Visual C# Express* detecta las características de cada uno de los elementos de la lista.

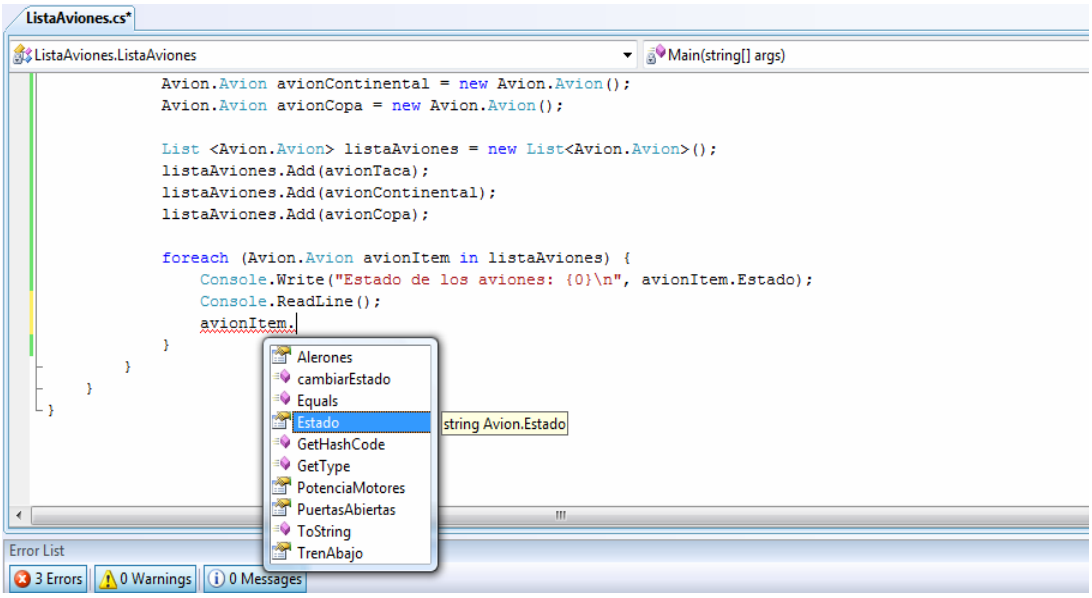

Figura 30: *Intellisense.*

La clase *List* es una de las colecciones genéricas más comunes y útiles del sistema de nombres *System.Collections.Generic*. Sirve para implementar cualquier lista de objetos que se considere necesaria en los programas. Su clase "hermana", *LinkedList*, crea una lista doblemente enlazada, como las que se conocen en el curso de Estructuras de Datos. Estas pueden ser recorridas tanto para adelante como para atrás. La que se implementa en el ejemplo 1, únicamente puede recorrerse hacia adelante. Observe la figura 26.2 del libro, página 1020 para conocer otras clases genéricas de colección con las que cuenta el *Framework .NET*.

Siguiendo con el ejemplo de la lista de aviones, y con el fin de trabajar con las colecciones no genéricas, se puede utilizar una tabla *hash* para controlar el conjunto de aviones que se encuentran en el espacio aéreo cercano. Las tablas *hash* utilizan una llave para clasificar los

elementos que se agregan a la misma. Cada llave debe ser distinta, para ubicar el elemento en un espacio distinto de la tabla.

La figura 31 brinda un ejemplo gráfico de cómo "visualizar" una tabla *hash* en memoria.

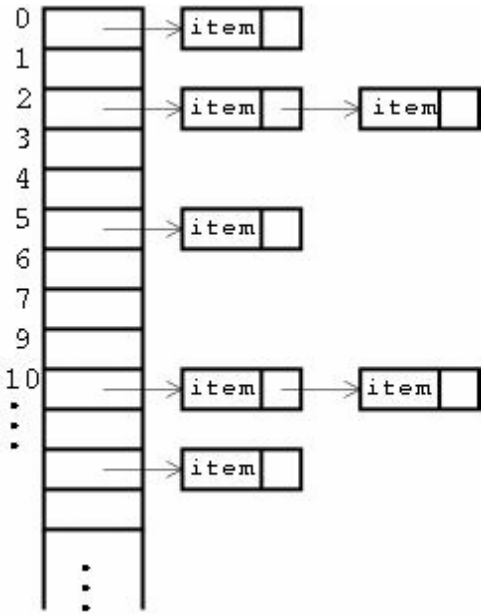

Figura 31: Representación visual de una tabla *hash.*
Fuente: <http://www.isr.umd.edu/~austin/ence200.d/java-examples.d>.

Cada uno de los ítemes de la tabla es un objeto de la clase Avión. Estos son ubicados en un espacio de memoria utilizando una función de *hash*, que recibe un valor asociado al objeto por agregar, que es procesado por esta función, y produce un resultado que se convierte en la llave que ubica a la instancia en un espacio de memoria. Se puede ver la implementación de una tabla en el ejemplo 2.

Ejemplo 2

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

```
using Avion;

namespace HashAviones
{
    class HashAviones
    {
        static void Main(string[] args)
        {
            Hashtable hashAviones = new Hashtable();
            Avion.Avion avionTaca = new Avion.Avion("TAU-456");
            Avion.Avion avionContinental = new Avion.Avion("ABC-890");
            Avion.Avion avionCopa = new Avion.Avion("TIA-123");

            hashAviones.Add(avionTaca.Placa, avionTaca);
            hashAviones.Add(avionContinental.Placa, avionContinental);
            hashAviones.Add(avionCopa.Placa, avionCopa);

            //Recorremos la tabla Hash
            foreach (object llave in hashAviones.Keys) {
                Avion.Avion itemAvion = (Avion.Avion)hashAviones[llave];
                Console.Write("Estado de los aviones recorrido en foreach:
{0}\n", itemAvion.Estado);
                Console.ReadLine();
            }
        }
    }
}
```

Observe que al ser una colección no genérica, no se especifican los tipos de objetos que contiene la tabla. Cuando se accede a ellos, deben ser convertidos de la clase *object* a la clase Avión. Adicionalmente, note cómo el objeto se extrae de la lista con los operadores de paréntesis cuadrados, como los elementos de un vector, utilizando como índice la llave del que se desea extraer de la tabla. Para el ejemplo 2, se utilizó cada uno de los valores de la colección *Keys* de la tabla *hash*.

Escoger las colecciones adecuadas para manipular y guardar los datos es una buena práctica de programación. Una correcta utilización de estas le permite crear *software* más eficiente en términos de memoria, espacio y tiempo de respuesta. Aunque en las aplicaciones de pequeña escala que se desarrollan en el curso estas variables no son tan importantes, cuando se realizan grandes desarrollos a nivel comercial, una diferencia

notable en tiempos de respuesta, espacio en disco o utilización de memoria en un servidor, causa que el producto sea más atractivo a los usuarios finales.

 Utilice las colecciones de datos genéricas o no genéricas en lugar de crear vectores de objetos. Estas vienen optimizadas de fábrica e incorporan funciones de alto nivel que no posee un vector de objetos. Adicionalmente, crecen de forma dinámica y por lo tanto utilizan mejor la memoria de la computadora.

Ejercicios de autoevaluación 3.1

Conteste las siguientes preguntas y realice el ejercicio que se sugiere.

1. ¿Cuál es la diferencia entre una colección genérica y una no genérica?

_____

_____

_____

_____

_____

2. Según el tipo de datos y el proceso, coloque la colección genérica o no genérica que considere más adecuada de utilizar. Piense en el proyecto de simulación del aeropuerto que se ha desarrollado durante el curso.

Cuadro 2
Colecciones genéricas y no genéricas

Datos y proceso	Colección sugerida
Manejo de las puertas del aeropuerto para descargar pasajeros.	
Manejo de las solicitudes de los aviones para despegar o aterrizar.	
Control de los aviones que han utilizado el aeropuerto en los últimos 30 minutos.	
Control de los aviones de cada una de las aerolíneas que visitan el aeropuerto.	

3. ¿Cómo funciona la estructura de control *foreach* cuando es utilizada en el contexto de las colecciones?

_____

_____

_____

_____

_____

4. Realice en un proyecto de *Visual C#* el ejercicio de la figura 26.6 del libro. Cámbielo de forma tal que en lugar de utilizar una pila no genérica, utilice una genérica que contenga *Strings*. (El ejemplo se encuentra en el archivo *cap26.zip* que se provee en el *CD-ROM* del libro de texto).

Recuerde que las respuestas a estos ejercicios se encuentran en la plataforma virtual del curso.

Laboratorios

En esta sección, los laboratorios corresponden uno a cada tema estudiado. En el primero se realizará un caso de conexión hacia bases de datos de pruebas. Es necesario que tenga instalado el "*SQL 2008 Express Edition with Tools*" en su máquina. Se resuelve en esta guía de estudio.

En el segundo laboratorio, se implementará una colección de tipo *LinkedList* en un proyecto de consola, utilizando la clase avión como base para la lista. Adicionalmente se implementarán métodos a esta colección con el fin de facilitar su uso en el proyecto programado final. Este queda como ejercicio adicional para ser realizado en casa o en la tercera tutoría.

Laboratorio 3.1

Realice los siguientes pasos. Guarde la solución de este laboratorio. Le será de ayuda para el segundo proyecto del curso.

1. Cree una nueva solución en *Visual C#* y nómbrela como AeropuertoBD.
2. Agregue la clase Avión que ha creado en los ejercicios anteriores y cree una referencia a ella en su nuevo proyecto.
    a. Para realizar una referencia, primero copie la carpeta del proyecto Avión dentro del *folder* de la solución AeropuertoBD.
    b. Agregue el proyecto a la solución, haciendo *clic* derecho sobre la solución y presionando sobre *Add*, luego *Existing Project*, como se muestra en la figura 32.

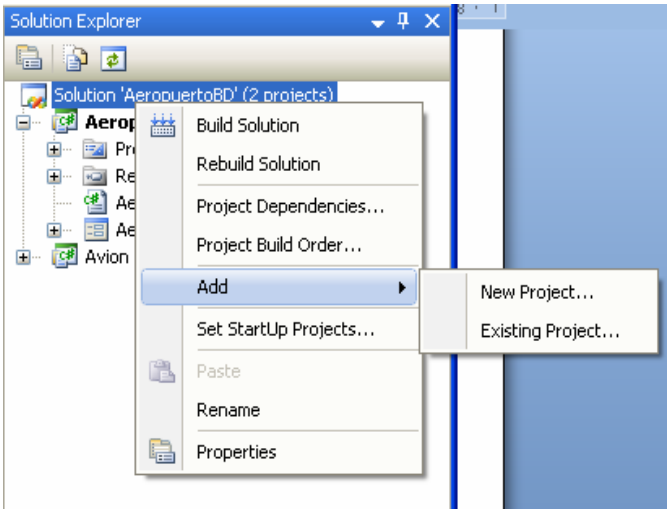

Figura 32: Agregar un proyecto a una solución.

c. Cuando lo haya agregado, cree una referencia al proyecto haciendo *clic* sobre el ícono de Referencias del proyecto AeropuertoBD, como se muestra en la figura 33.

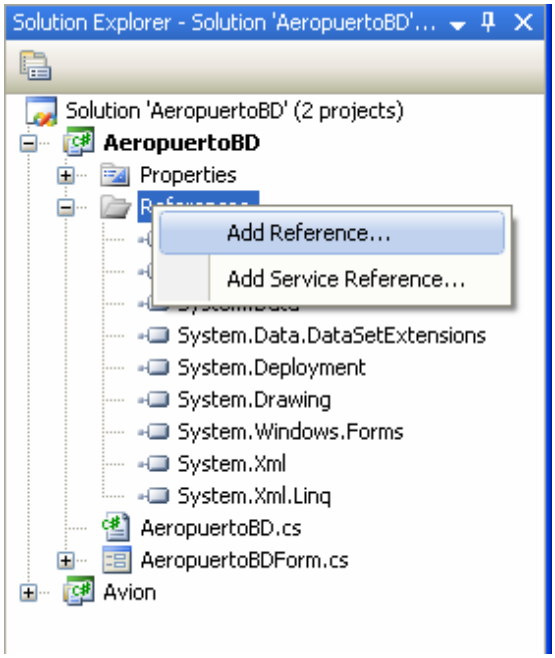

Figura 33: Agregar una referencia a un proyecto.

d. Escoja la pestaña de *Projects*, luego escoja Avión y, finalmente, haga *clic* sobre *OK*, como se muestra en la figura 34.

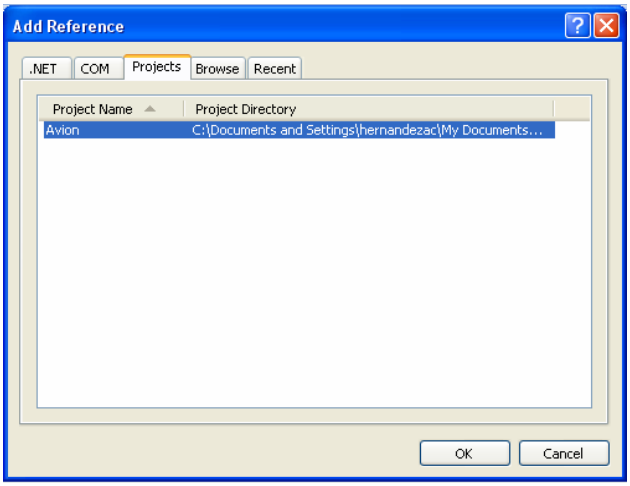

Figura 34: Agregar una referencia al avión.

3. Diseñe una nueva ventana que represente las acciones de un avión. Agregue una etiqueta para indicar el estado actual y una etiqueta para mostrar el número de placa y la aerolínea. En la figura 35 se muestra un ejemplo de una pantalla.

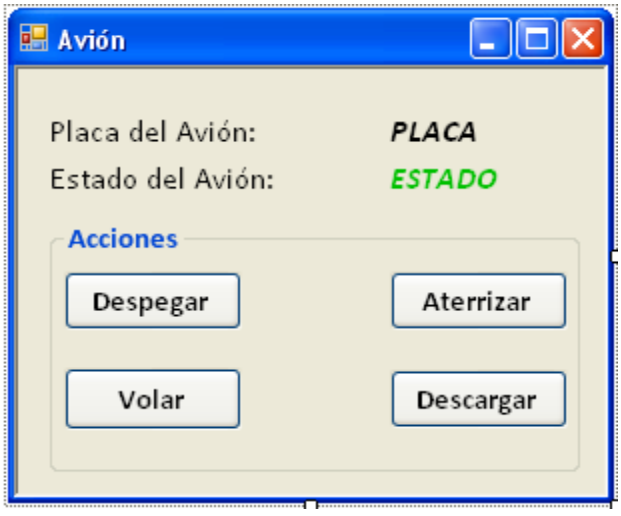

Figura 35: Una pantalla para el avión.

4. Ahora diseñe una pequeña pantalla para capturar la placa y la aerolínea del avión cuando se crea una nueva ventana. Los valores capturados en este formulario se

deben enviar a la ventana del avión. Para este ejemplo hemos nombrado este formulario como *CapturaPlacaForm*. Se sugiere un diseño como el que se muestra en la figura 36:

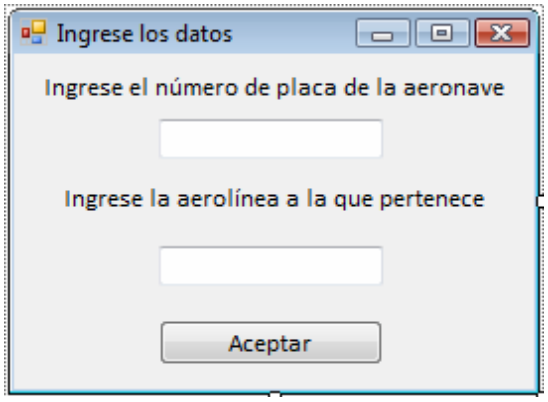

Figura 36: El formulario *CapturaPlacaForm*.

5. Diseñe adicionalmente un contenedor *MDI*, como el de la figura 37, para que los dos formularios anteriores solamente puedan ser visualizados dentro de éste. Cree un menú en ese formulario, utilizando un objeto *MenuStrip*, como se realizó en la sección 1 de esta guía.

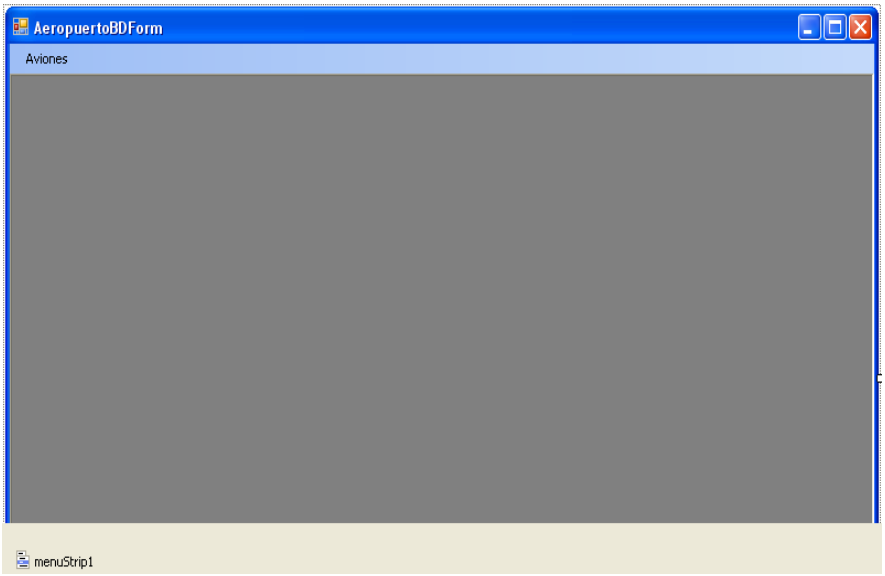

Figura 37: Contenedor *MDI* para los aviones.

6. En las opciones del menú, agregue una para crear un nuevo avión. Cuando esta opción se seleccione, debe obtener el número de placa del avión por crear y luego desplegarlo. Limite el número de aviones simultáneos a 5. Observe el siguiente bloque de código que programa esa funcionalidad:

```
private void crearNuevoAviónToolStripMenuItem_Click(object sender,
EventArgs e)
{
    if (this.MdiChildren.Length < 5)
    {
        CapturaPlacaForm placa = new CapturaPlacaForm();
        placa.MdiParent = this;
        placa.Show();
    }
    else
    {
        MessageBox.Show("El número máximo de aviones simultáneos es 5.",
"Limite alcanzado", MessageBoxButtons.OK);
    }
}
```

7. En el evento del botón aceptar del formulario *CapturaPlacaForm,* cree una nueva ventana de avión, enlácela con el formulario *MDI* padre y finalmente disponga de la ventana de capturar placa. Observe el bloque de código siguiente donde se implementa esta funcionalidad:

```
private void Aceptar_Click(object sender, EventArgs e)
{
    if (Placa.Text == "")
    {
        MessageBox.Show("Por favor ingrese todos los datos", "Datos",
MessageBoxButtons.OK);
    }
    else
    {
        AvionBDForm frmAvion = new AvionBDForm(Placa.Text,
Aerolinea.Text);
        frmAvion.MdiParent = AeropuertoBDForm.ActiveForm;
        this.Dispose();
        frmAvion.Show();
    }
}
```

8. Descargue de la plataforma virtual el *script* de la base de datos Aeropuerto. Cree una nueva base de datos en *SQL 2008* y cargue el *script* en ella. Observe la figura 38 con el diagrama.

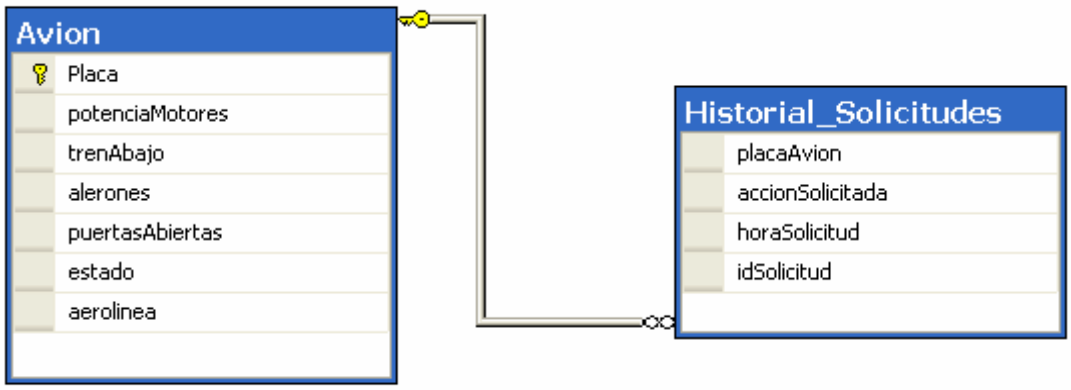

Figura 38: Diagrama de la base de datos Aeropuerto.

9. A continuación observe el *script* de la base de datos Aeropuerto.

```
USE [Aeropuerto]
GO
/** Object:  Table [dbo].[Avion]    Script Date: 05/27/2009 15:38:35 **/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Avion](
      [Placa] [nvarchar](50) NOT NULL,
      [potenciaMotores] [int] NOT NULL,
      [trenAbajo] [smallint] NOT NULL,
      [alerones] [nvarchar](50) NOT NULL,
      [puertasAbiertas] [smallint] NOT NULL,
      [estado] [nvarchar](50) NOT NULL,
      [aerolinea] [nvarchar](50) NULL,
 CONSTRAINT [PK_Avion] PRIMARY KEY CLUSTERED
(
      [Placa] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY =
OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
GO
/****** Object:  Table [dbo].[Historial_Solicitudes]    Script Date:
05/27/2009 15:38:35 ******/
SET ANSI_NULLS ON
GO
```

```
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Historial_Solicitudes](
      [placaAvion] [nvarchar](50) NOT NULL,
      [accionSolicitada] [nvarchar](50) NOT NULL,
      [horaSolicitud] [datetime] NOT NULL,
      [idSolicitud] [int] IDENTITY(1,1) NOT NULL
) ON [PRIMARY]
GO
/****** Object:  ForeignKey [FK_Historial_Aviones_Avion]    Script Date:
05/27/2009 15:38:35 ******/
ALTER TABLE [dbo].[Historial_Solicitudes]  WITH CHECK ADD  CONSTRAINT
[FK_Historial_Aviones_Avion] FOREIGN KEY([placaAvion])
REFERENCES [dbo].[Avion] ([Placa])
ON UPDATE CASCADE
ON DELETE CASCADE
GO
ALTER TABLE [dbo].[Historial_Solicitudes] CHECK CONSTRAINT
[FK_Historial_Aviones_Avion]
GO
```

10. En el constructor del formulario *AvionBDForm*, cada vez que se crea un avión nuevo, debe insertar un registro en la tabla Avión de la base de datos. Para ello utilice los objetos *ADO.NET* de C#. El siguiente fragmento de código implementa dicha solución.

```
public AvionBDForm(String Matricula, String Airline)
{
    InitializeComponent();
    aeronave = new Avion.Avion();
    placaAvion.Text = Matricula;
    estadoAvion.Text = aeronave.Estado;
    aerolineaAvion.Text = Airline;
    // Se agrega la funcionalidad para insertar un registro en la base
de datos para el nuevo avion.
    SqlConnectionStringBuilder bldr = new SqlConnectionStringBuilder();
    bldr.DataSource = ".\\SQLExpress";
    bldr.InitialCatalog = "Aeropuerto";
    bldr.UserID = "usrAeropuerto";
    bldr.Password = "*usrAero*";
    SqlConnection conx = new SqlConnection(bldr.ConnectionString);
    String SQL = "INSERT into AVION (Placa, potenciaMotores, trenAbajo,
alerones, puertasAbiertas, estado, aerolinea)";
    SQL += " VALUES ('" + Matricula + "', " + aeronave.PotenciaMotores
+ ", " + (aeronave.TrenAbajo ? 1 : 0) + ", '" + aeronave.Alerones + "',
" + (aeronave.PuertasAbiertas ? 1 : 0) + ", '" + aeronave.Estado + "',
'" + Airline + "')";
try
{
   conx.Open();
   SqlCommand comandoInsertar = new SqlCommand(SQL, conx);
   comandoInsertar.ExecuteNonQuery();
```

```
    }
    catch (Exception err) {
        MessageBox.Show(err.ToString());
    }
    conx.Close();
}
```

11. Del fragmento de código, observe los siguientes puntos:
    a. Se creó un nuevo constructor para el formulario, agregando dos parámetros al mismo, así como adicionando la funcionalidad de escribir en la base de datos.
    b. Se construye la cadena de conexión utilizando un objeto *SqlConnectionStringBuilder.* Cuando se tiene poco conocimiento de cómo construir cadenas de conexión a una base de datos, este *"Code Snippet"* es de gran utilidad. Observe cómo luego simplemente se asigna el atributo *ConnectionString* del objeto *bldr* al *SqlConnection* cuando se crea la conexión.
    c. Se crea una sentencia de *SQL* basada en los atributos del nuevo objeto Avión que deben escribirse a la base de datos.
    d. Se utiliza el operador **"?":** Este valida una condición *booleana* "en línea" y contesta con el primer valor después del operador en caso de ser verdadero y con el segundo si es falso. Recuerde que los valores que utilizan este operador son enteros en la base de datos.
    e. Se realiza una operación de base de datos mediante la función *ExecuteNonQuery*(). Esta función se utiliza para ejecutar una sentencia de *SQL* que no retorna registros (un *SELECT*, por ejemplo), o para ejecutar procedimientos almacenados.

12. A continuación, programe cada uno de los eventos de los botones del avión. Cada vez que un avión ejecute una solicitud válida, debe realizar dos acciones en la base de datos.
    a. El avión debe cambiar todos los valores de la tabla Avión para colocar sus atributos como se encuentran luego del cambio.

b. El avión debe registrar un evento en la tabla *Historial_Solicitudes*, con el fin de dejar una bitácora de todas las solicitudes que realizan todos los aviones.

13. Las siguientes instrucciones realizan las acciones del punto anterior para el evento despegar. Todos los demás realizan el mismo proceso en este paso.

```
private void Despegar_Click(object sender, EventArgs e)
{
    if (aeronave.cambiarEstado(1))
    {
    //Se crean las dos sentencias de SQL para escribir en la BD.
        SQL = "UPDATE Avion SET potenciaMotores = " +
aeronave.PotenciaMotores + ", trenAbajo = " + (aeronave.TrenAbajo ? 1 : 0)
+ ", alerones = '" + aeronave.Alerones + "', puertasAbiertas = " +
(aeronave.PuertasAbiertas ? 1 : 0) + ", estado = '" + aeronave.Estado +
"' WHERE Placa = '" + placaAvion.Text + "'" ;

        SQLBitacora = "INSERT INTO Historial_Solicitudes (placaAvion,
accionSolicitada, horaSolicitud)" + "VALUES ('" + placaAvion.Text + "',
'Despegar', '" + DateTime.Now.ToString("yyyy-MMM-dd hh:mm:ss") + "')";

        estadoAvion.Text = aeronave.Estado;
        try
        {
            conx.Open();
            SqlCommand comandoCambiarEstado = new SqlCommand(SQL, conx);
            comandoCambiarEstado.ExecuteNonQuery();
            SqlCommand comandoBitacora = new SqlCommand(SQLBitacora,
            conx);
            comandoBitacora.ExecuteNonQuery();
            conx.Close();
        }
        catch (SqlException sqlErr){
            MessageBox.Show(sqlErr.ToString());
        }
    }
    else
    {
        MessageBox.Show("El cambio de estado no es coherente. Intente de
nuevo", "Cambio Incorrecto", MessageBoxButtons.OK);
    }
}
```

14. Siguiendo los pasos anteriores, todos los aviones escribirán sus datos y solicitudes en la base de datos. Si lo desea, puede agregar funciones para controlar que los aviones no se puedan cerrar sin antes aterrizar y descargar.

Laboratorio 3.2

1. Cree una nueva solución en *Visual C# Express* y nómbrela como *LinkedListAvion*.

2. Programe las funciones de la lista de forma tal que sea una lista de aviones.

3. Cree una nueva aplicación de consola que realice lo siguiente:
    a. Que capture un número ingresado por el usuario que represente la cantidad de objetos de tipo Avión que debe tener la lista.
    b. Que cree los objetos generando una placa para el avión de manera aleatoria.
    c. Que presente un menú donde se pueda recorrer la lista hacia atrás y hacia adelante y terminar el programa.
    d. Cuando el programa termine, primero debe disponer de la memoria asignada a la lista.

4. Este ejercicio será realizado en la tercera tutoría del curso. Recuerde que puede encontrar una solución programada a este ejercicio en la plataforma virtual, en caso de que no pueda asistir a la tutoría.

5. No olvide documentar sus aprendizajes de los dos ejercicios en sus reportes de laboratorio.

Si no conoce el funcionamiento de las *LinkedList*, puede realizar el ejercicio de la figura 26.9 del libro de texto. Adicionalmente, recuerde que cuenta con la biblioteca *MSDN* para complementar sus conocimientos en cualquier objeto del *Framework* de *.NET*. Para consultar los detalles del objeto *LinkedList*, puede visitar el siguiente enlace en Internet: <http://msdn.microsoft.com/en-us/library/he2s3bh7.aspx>.

Glosario

ADO. Siglas para *ActiveX Data Objects*, objetos de datos activos.

API. Siglas para *Application Program Interface*, interfaz de programación de aplicaciones.

intellisense. Sistema de ayuda automática del ambiente de programación *.NET*, que sugiere al programador los objetos, funciones, atributos y ayudas relacionadas con el lenguaje con el que se encuentra trabajando.

OLEDB. Siglas para *Object Linking and Embedding for Databases*, enlace e incrustación de objetos para bases de datos. Tecnología desarrollada por Microsoft para tener acceso a diferentes fuentes de información o bases de datos.

SQL. Siglas para *Structured Query Language*, lenguaje de consultas estructurado.

V. Cuarta sección (Cuarta tutoría) Redes

Es indiscutible que las redes de computadoras son el corazón de la informática moderna. La interconexión e integración de servicios han permeado todos los ambientes, modernizando y cambiando la forma en que las personas y servicios se comunican.

En esta evolución, los lenguajes de programación han jugado un papel muy importante. Las nuevas aplicaciones poseen en su mayoría funciones de red, y acceden fuentes de datos que provienen de aplicaciones remotas, así como de servicios *Web* y bases de datos externas. En este curso se estudian las funciones, objetos e instrucciones necesarias para incluir la comunicación en red como parte de aplicaciones avanzadas.

Tema 7: Redes. *Sockets* basados en flujos y datagramas

Antes de iniciar el desarrollo de aplicaciones que utilizan objetos y funciones de red, es importante aclarar algunos de los conceptos más comunes e importantes que serán mencionados durante el transcurso de esta sección de la guía, y que deben ser de su conocimiento, ya que forman parte de la teoría básica de redes y telecomunicaciones.

- **Paquete:** Un paquete es un conjunto de datos encapsulados que viajan a través de la red y que contienen información relevante de una conexión específica.

- **Protocolo:** Un protocolo es un conjunto de reglas que definen la manera en que se realiza el intercambio de datos cuando se establece una comunicación entre distintos sistemas, sin importar el sistema operativo. Definen un formato para el intercambio de mensajes y la transmisión de los mismos a través del medio.

- **Protocolo orientado a la conexión:** Un protocolo se considera orientado a la conexión cuando las partes que participan en él, el cliente y el servidor, efectúan una conexión o una validación de presencia uno contra el otro. Adicionalmente, esa conexión permanece abierta mientras ambas partes se encuentren en comunicación, y el intercambio de paquetes es constante entre ellos. Además, realizan verificaciones constantes para validar que un paquete que fue enviado desde el emisor, haya sido recibido de manera íntegra en el receptor. Son el equivalente a una llamada telefónica, donde ambos interlocutores permanecen conectados al teléfono mientras dure la conversación. El más difundido y común en la actualidad es el *Transmission Control Protocol*, o *TCP*.

- **Protocolo no orientado a la conexión:** Un protocolo no orientado a la conexión es aquel que no efectúa una validación de presencia entre las partes que efectúan la comunicación. Trabajan bajo la premisa de entrega con el mejor esfuerzo, dado que no validan si los paquetes alcanzaron su destino, sino que simplemente envían tramas o datagramas hacia un destino determinado. Son el equivalente a un servicio de correo postal, donde el cartero deja la carta en el buzón, sin verificar si el destinatario se encuentra o no en su casa. El más utilizado por las aplicaciones modernas es el *User Datagram Protocol*, o *UDP*.

- ***Socket:*** Un *socket* es similar a un canal de comunicación que utilizan dos aplicaciones para transmitir datos. Se compone de un número de puerto, un protocolo y direcciones *IP*.

- **Protocolo de Internet:** *IP*, siglas para *Internet Protocol*. Es el protocolo de red más importante del mundo. Es un protocolo enrutado, no orientado a la conexión y que utiliza la técnica de entrega por mejor esfuerzo. Su principal componente son los paquetes *IP*, que son conjuntos de datos encapsulados dentro de un encabezado y que poseen información específica para llevar a cabo el transporte de los mensajes a través de la red de su origen al destino.

- **Modelo Cliente/Servidor:** Las aplicaciones que utilizan las redes para transmitir información, pueden emplear varios modelos de comunicación. El Cliente/Servidor es uno de ellos. En este patrón, las aplicaciones cliente establecen contacto con una aplicación central conocida como servidor. Este contiene la lógica compleja y de negocio, así como el contacto con la base de datos. Los clientes extraen la información del servidor, realizan operaciones lógicas y aritméticas con ella y vuelven a conectarse para depositarla. La mayoría de las aplicaciones cliente/servidor utilizan protocolos orientados a la conexión. Algunos ejemplos son el correo electrónico, el protocolo *Web HTTP* y el protocolo *DNS*.

Establecimiento de comunicaciones e interacción entre aplicaciones

Cuando dos aplicaciones desean comunicarse a través de la red, deben establecer un canal de comunicación ya sea utilizando flujos de datos con *Sockets TCP* o con datagramas. En el caso de *TCP*, un servidor espera una solicitud de uno o varios clientes y las procesa luego de concertar con el cliente los puertos por donde dicho flujo de datos será realizado.

La figura 39 muestra con un diagrama el establecimiento de una comunicación *TCP*. Recuerde que las comunicaciones con *UDP* no establecen un canal de comunicación, solamente envían datagramas desde y hacia un puerto determinado.

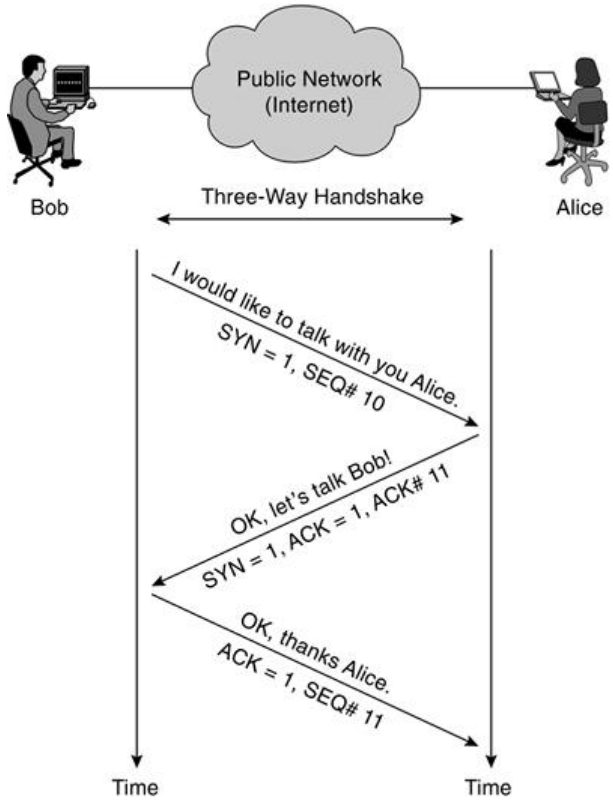

Figura 39: Establecimiento de un canal de comunicación *TCP.*
Fuente: <http://mudji.net/press/wp-content/uploads/2006/11/3way.JPG>.

Implementación de un servidor *TCP* básico

Como se menciona en la sección 23.4 del libro, página 919, son necesarios cinco pasos para establecer un servidor *TCP* utilizando el lenguaje de programación C#:

1. Crear una instancia de un objeto de la clase *TcpListener* del espacio de nombres *System.Net.Sockets,* pasando como parámetros del constructor de la clase, la dirección *IP* sobre la que el servidor debe establecerse, así como el número de puerto sobre el que escuchará el *socket*. Observe en la página 920 del libro de texto la forma de realizar dicha creación.
2. Llamar al método *Start* del objeto *TcpListener*, que inicia la escucha del *socket* por solicitudes de conexión.

3. Crear una instancia de un objeto *NetworkStream*, que utiliza el objeto *socket* para enviar y recibir datos entre las aplicaciones.
4. Procesar las cadenas de datos utilizando objetos *BinaryReader* y *BinaryWriter* para leer y escribir los flujos desde y hacia las aplicaciones.
5. Terminar la conexión. Llamar al método *Close* de los objetos anteriores. Luego, llamar a los métodos *Shutdown* y *Close* de la clase *Socket*.

Observe el ejemplo 1 que muestra una codificación básica para un servidor *TCP* con subprocesamiento múltiple, donde se cumplen los dos primeros pasos de la lista anterior.

Ejemplo 1

```
using System;
using System.Text;
using System.Net.Sockets;
using System.Threading;
using System.Net;

namespace TCPServerBasic
{
  class Server
  {
    private TcpListener tcpListener;
    private Thread listenThread;

    public Server()
    {
      this.tcpListener = new TcpListener(IPAddress.Any, 30000);
      this.listenThread = new Thread(new ThreadStart(ListenForClients));
      this.listenThread.Start();
    }
  }
}
```

Como pudo notar, estas líneas de código son apenas el constructor de la clase, sin las demás funciones para atender clientes. Preste atención a la forma en que se inicia el subprocesamiento múltiple, creando un objeto de tipo *Thread* y enviando como parámetro en su constructor a la función *ListenForClients*. En el ejemplo 2, se muestra la codificación de esta función.

Ejemplo 2

```
private void ListenForClients()
{
  this.tcpListener.Start();
  while (true)
  {
    //Se bloquea hasta que un cliente se haya conectado al servidor
    TcpClient client = this.tcpListener.AcceptTcpClient();
    /*Se crea un nuevo hilo para manejar la comunicación con los clientes
que se conectan al servidor*/
    Thread clientThread = new Thread(new
ParameterizedThreadStart(HandleClientComm));
    clientThread.Start(client);
  }
}
```

La función anterior es muy simple. Primero inicia el *TcpListener* de la clase con su método *Start.* Luego ingresa a un ciclo infinito para aceptar conexiones. Como se menciona en la documentación interna, la función *AcceptTcpClient* se bloquea hasta que un cliente se ha conectado. En ese momento se crea un nuevo hilo de ejecución (subproceso) para manejar la comunicación con el cliente que recién ha establecido comunicación. Este nuevo hilo se instancia utilizando el método *ParametrizedThreadStart* para enviar al constructor del subproceso el objeto *TcpClient* que se retorna como producto de la llamada a *AcceptTcpClient.* De esta forma se implementa el subprocesamiento en el servidor.

La función que se utiliza para manejar la comunicación con los clientes es *HandleClientComm.* Esta es responsable de leer los datos que provienen del cliente. El ejemplo 3 muestra el código de esta función, donde se cumplen los tres últimos pasos de la lista de la página 96 de esta guía.

Ejemplo 3

```
private void HandleClientComm(object client)
{
  TcpClient tcpClient = (TcpClient)client;
  NetworkStream clientStream = tcpClient.GetStream();

  byte[] message = new byte[4096];
  int bytesRead;

  while (true)
  {
    bytesRead = 0;

    try
    {
      /*Esta sección se bloquea hasta que el cliente envíe un
      mensaje*/
      bytesRead = clientStream.Read(message, 0, 4096);
    }
    catch
    {
      //Ocurrió un error en el socket
      break;
    }

    if (bytesRead == 0)
    {
      //El cliente se desconectó del servidor
      break;
    }

    //Mensaje recibido correctamente
    ASCIIEncoding encoder = new ASCIIEncoding();
    System.Diagnostics.Debug.WriteLine(encoder.GetString(message, 0,
bytesRead));
  }

  tcpClient.Close();
}
```

Lo primero que se realiza es convertir el objeto cliente en un *TcpClient*. De esta forma se recupera el objeto que se envió desde el constructor del subproceso en la función anterior. Seguidamente, obtenemos un objeto de tipo *NetworkStream* desde el cliente *TCP*, que se utiliza para leer los datos desde el cliente conectado. La lectura se realiza dentro de un ciclo infinito. El llamado al método *Read* se bloquea hasta que se reciba un mensaje. Si la

lectura retorna 0 *bytes*, es porque el cliente se ha desconectado. De otra forma, el mensaje ha sido recibido correctamente, y por lo tanto se convierte el arreglo de *bytes* a un *String*. En el ejemplo 3 esta cadena se escribe en la consola de depuración, pero se puede realizar cualquier otra acción con los datos. Finalmente, si el *socket* tiene algún tipo de error o simplemente el cliente se desconecta, se debe realizar un llamado a la función *Close* del cliente *TCP* para liberar cualquier recurso que esté en uso en ese momento.

Así de simple es crear un servidor *TCP* con subprocesamiento múltiple. Como complemento, puede agregar algunas líneas para realizar cosas más interesantes con la cadena de datos recibida desde el cliente. El ejemplo 4 muestra las líneas de código necesarias para realizar dicho proceso. Estas líneas pueden insertarse en el código del ejemplo 3, si lo desea.

Ejemplo 4

```
NetworkStream clientStream = tcpClient.GetStream();
ASCIIEncoding encoder = new ASCIIEncoding();
byte[] buffer = encoder.GetBytes("Hello Client!");

clientStream.Write(buffer, 0 , buffer.Length);
clientStream.Flush();
```

Enviar datos a los clientes conectados es sumamente sencillo. Todo lo que se debe hacer es realizar un llamado a la función *Write* del objeto *NetworStream* y colocar como parámetro el arreglo de *bytes* que desea enviar. Luego efectúe un llamado al método *Flush*, para limpiar la cola de transmisión. Su servidor *TCP* está ahora completo. Como puede notar, programar la interconexión es rápido. Lo que toma más tiempo es definir qué protocolo de envío de datos utilizarán las aplicaciones para entender las instrucciones que envían y reciben. En el laboratorio 4.1, se definirá un protocolo simple de transmisión de instrucciones para comunicar a los aviones con el controlador aéreo.

Implementación de un cliente *TCP* básico

En contraparte, son necesarios los 4 pasos siguientes para crear un servidor *TCP* simple utilizando el lenguaje de programación C#, como se menciona en la sección 23.5 del libro, página 921:

1. Crear una instancia de un objeto de la clase *TcpClient.* Inmediatamente después, llamar a la función *Connect* de dicho objeto, colocando como parámetros de la función la dirección *IP* del servidor con el que se desea establecer la conexión, así como el número de puerto por el que el *socket* del servidor está escuchando.

2. Utilizar el método *GetStream* del cliente y obtener un objeto de clase *NetworkStream* para escribir y leer datos del *socket*. Adicionalmente, se utiliza también dos objetos de tipo B*inaryReader y BinaryWriter,* para enviar y recibir información desde y hacia el servidor.

3. Procesar las cadenas de datos con los métodos *Write* y *ReadString* de los objetos declarados anteriormente.

4. Cerrar la transmisión. Para ello se realizan llamadas a la función *Close* de todos los objetos creados.

El ejemplo 5 muestra las instrucciones básicas para crear un cliente *TCP* básico que tenga la capacidad de conectarse al servidor que se muestra en los ejemplos 1 a 4 de esta sección de la guía. Nótese que este cliente no utiliza objetos *BinaryReader* ni *BinaryWriter* para escribir datos hacia el servidor, como sí lo hacen los ejemplos del libro de texto.

Ejemplo 5

```
TcpClient client = new TcpClient();

IPEndPoint serverEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"),
30000);

client.Connect(serverEndPoint);

NetworkStream clientStream = client.GetStream();

ASCIIEncoding encoder = new ASCIIEncoding();
byte[] buffer = encoder.GetBytes("Hello Server!");

clientStream.Write(buffer, 0 , buffer.Length);
clientStream.Flush();
```

Lo primero que se debe realizar es conectar al cliente con el servidor. Para ello se utiliza el método *Connect* del *TcpClient*. Este método requiere de un objeto tipo *IPEndPoint* para realizarla. En este caso se declara uno hacia *localhost* por el puerto 30000, luego se envía una cadena hacia el servidor *"Hello Server"*.

Un detalle importante de recordar es que una escritura en el cliente no necesariamente equivale a una lectura en el servidor o viceversa. Por ejemplo, su cliente puede enviar 20 *bytes* hacia el servidor, pero este podría no recibir los 20 *bytes* la primera vez que ejecuta una lectura sobre el *socket*. Si se utiliza *TCP*, es casi una garantía que se recibirán todos pero podría tomar más de una lectura del *socket*. Tome en cuenta esta recomendación cuando diseñe su protocolo de comunicación de instrucciones.

Si desea consultar ejemplos adicionales sobre la creación de servidores y clientes *TCP*, puede visitar el sitio de *java2us* en:
<http://www.java2s.com/Code/CSharp/Network/CatalogNetwork.htm>.


Tenga cuidado al asignar puertos cuando se instancian servidores *TCP* o *UDP*. Existen estándares internacionales que indican qué aplicaciones utilizan puertos específicos. Estos se conocen como "bien conocidos", y van desde el 0 al 1 024. Se encuentran reservados para servicios privilegiados. Siempre debe respetar este estándar a la hora de utilizar *sockets* en sus aplicaciones.


Los puertos del 1 025 al 65 536 son de libre utilización, pero también existen aplicaciones conocidas que usan algunos de estos y que también deben respetarse por cuestiones de estándar internacional. Una buena práctica es utilizar puertos muy altos, superiores al 30 000.


Nunca asigne un número de puerto mayor a 65 536. Por limitación del protocolo *TCP*, esto no puede realizarse. Asignar uno mayor a un servidor *TCP* producirá un error en su aplicación.


La asignación de puertos conocidos está controlada por la *IANA* (*Internet Assigned Numbers Authority*). Utilice el siguiente enlace como referencia para evitar conflictos a la hora de utilizar algún puerto para sus aplicaciones: <http://www.iana.org/assignments/port-numbers>.

Práctica programada #4

Para practicar los conceptos aprendidos, ejecute los ejercicios de las figuras 23.1, 23.2, 23.3 y 23.4 del libro de texto, páginas 922 a 936.

1. Obtenga el archivo *cap23.zip* del *CD-ROM* que se provee con el libro. Descomprima el contenido en una carpeta en su disco duro.

2. Cree los proyectos para las figuras 23.1 y 23.2 en *Visual C#* y guárdelo en sus proyectos como *ServidorTCP* y *ClienteTCP* respectivamente.

3. Cree los proyectos para las figuras 23.3 y 23.4 en *Visual C#* y guárdelo en sus proyectos como *ServidorUDP* y *ClienteUDP* respectivamente.

4. Observe el cuadro 1. Presenta un resumen con las principales instrucciones de C# para establecer una comunicación cliente / servidor utilizando el protocolo *UDP*. Realice un diagrama similar con las instrucciones para realizar establecer una comunicación cliente / servidor con el protocolo *TCP*.

Cuadro 1
Instrucciones de C# para utilizar el protocolo *UDP* en aplicaciones cliente / servidor

Instrucción de C#	**Función**
using System.Net; *using System.Net.Sockets;*	Librerías que contienen los objetos necesarios para trabajar con aplicaciones en red.
private UdpClient cliente; *private IPEndPoint puntoRecepcion;*	Declaración de un cliente y de un punto de recepción *UDP*. Un punto de recepción es el equivalente a un *socket TCP*.
cliente = new UdpClient(50000); *puntoRecepcion = new IPEndPoint(new IPAddress(0), 0);*	Se instancia el punto de recepción y se indica al objeto *UDP* servidor el puerto por el que debe escuchar.
byte[] datos = cliente.Receive(ref puntoRecepcion);	Se leen los datos utilizando el cliente *UDP* y enviando como parámetro el punto de recepción.
System.Text.Encoding.ASCII.GetString(datos) ;	Se realiza la conversión de datos para poder ser interpretados como un *string*. Los datos son recibidos con formato de *byte*.
System.Text.Encoding.ASCII.GetBytes(paquete);	Se realiza la conversión de una cadena para poder ser interpretados como un *byte*. Los datos se deben enviar por el *socket* como *bytes*.
cliente.Send(datos, datos.Length, puntoRecepcion);	Se envían los datos recibidos de vuelta. La función *Send* retorna la cadena de texto a través del punto de recepción.

Aplicaciones en red con subprocesamiento múltiple

Ahora que ya se conocen las bases teóricas de las redes y su aplicación práctica en el lenguaje de programación C#, es posible diseñar programas complejos y poderosos. Diariamente en la constante interacción con el *software* y especialmente con Internet, se está en contacto con aplicaciones que combinan muchos de los conceptos aprendidos en este curso, pero particularmente con dos en especial: El subprocesamiento y las comunicaciones en red.

Un servidor *Web* es un *software* que lleva esta expresión al máximo. Es un programa que crea un subproceso de atención para cada cliente, al mismo tiempo que se comunica con éste a través de *sockets TCP*. Un servidor de correo, un motor de base de datos, un servicio de mensajería instantánea y un servidor *proxy*, son ejemplos típicos de aplicaciones cotidianas que también utilizan este modelo.

En la sección 23.8 del libro, páginas de la 938 a 948 se muestra una aplicación de complejidad media que utiliza los conceptos de subprocesamiento, bloqueos y comunicación en red. La comprensión de ella será tomada en cuenta para la discusión de la actividad *Web* #2 del curso, por lo que su estudio es obligatorio. Descargue la solución programada de la plataforma virtual y con base en ella responda las preguntas que se le formularán en la actividad.

Si desea ampliar aún más sus conocimientos en este tema, existe otra aplicación con subprocesamiento y comunicación en red disponible en la plataforma *Web*. Descargue el archivo *ChatNetwork.rar* y descomprímalo en su computadora. Este proyecto consta de dos secciones: Un cliente y un servidor. Cree dos proyectos por separado y ejecútelos. La explicación del código y de la implementación se encuentra en la siguiente dirección: <http://www.codeproject.com/KB/IP/chatserver.aspx>.

Ejercicios de autoevaluación 4.1

Con base en lo estudiado en esta sección, responda las siguientes interrogantes.

A. Determine si la afirmación es falsa o verdadera. Si es falsa, explique por qué.

1. *UDP* es un protocolo orientado a la conexión.

---

2. Un proceso establece conexiones con otros procesos a través de *sockets*.

---

3. La transmisión de datagramas es confiable. Todos los paquetes llegan en secuencia a su destino.

---

4. La mayoría del tiempo, el protocolo *TCP* se prefiere sobre el protocolo *UDP.*

---

5. Una instancia de un objeto *TcpListener* puede escuchar por conexiones en más de un puerto a la vez.

---

6. Los paquetes enviados vía *UDP* se envían sólo una vez.

B. Complete los espacios en blanco con la o las palabras correctas.

1. Muchas de las clases utilizadas para realizar conexiones de red en C# se encuentran en los espacios de nombres _____ y _____.

2. La clase _____ se utiliza para transmisión rápida pero no confiable de datagramas.

3. Un objeto de clase _____ representa una dirección *IP.*

4. Los dos tipos de *sockets* vistos en esta sección de la guía son _____ y _____.

5. Las siglas *TCP* son el acrónimo de _____.

6. La clase _____ escucha por conexiones de los clientes.

7. La clase _____ se conecta a los servidores.

Laboratorios

Seguidamente se brindan las instrucciones para realizar los dos laboratorios de la cuarta sección de esta guía. Ambos se enfocan en el tema de comunicaciones a través de la red para aplicaciones programadas con C#. Para continuar con la temática que se ha trabajado durante el desarrollo del curso, en el primer laboratorio, realizará una modificación a una de las aplicaciones servidor con las que trabajó durante esta sección y la adecuará para que actúe como un escucha de peticiones de aviones. Adicionalmente deberá agregar funciones de red para sus "aviones" simulados, de forma tal que se comuniquen con el servidor. Este laboratorio se resuelve en esta guía de estudio.

En el segundo laboratorio, deberá programar una solución para comunicar dos clientes a través de un servidor con subprocesamiento, con la particularidad de que estos clientes cifran sus mensajes en código Morse cuando los envían hacia su contraparte. Este laboratorio queda como ejercicio adicional para ser realizado en casa o en la cuarta tutoría.

Laboratorio 4.1

En este laboratorio se realizarán dos actualizaciones a la aplicación de los aviones que se han simulado utilizando los conocimientos adquiridos en el curso.

1. Cree un nuevo proyecto en *Visual C# Express* y nómbrelo como *ServidorAviones*.

2. Agregue un nuevo formulario de Windows a su proyecto. Cambie las propiedades del formulario colocando los siguientes valores. Las figuras 40 a 42 muestran todas las propiedades que deben cambiarse.

a. Tamaño de 445 por 400 *píxeles*.

b. Título de ventana: "*Servidor Aviones*".

c. Nombre del formulario: *MainFormServido*r.

d. Tipo de borde de formulario: *FixedSingle*.

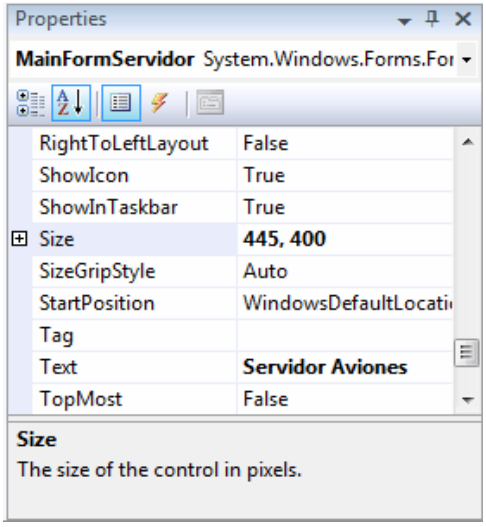

Figura 40: Propiedades tamaño y texto del formulario.

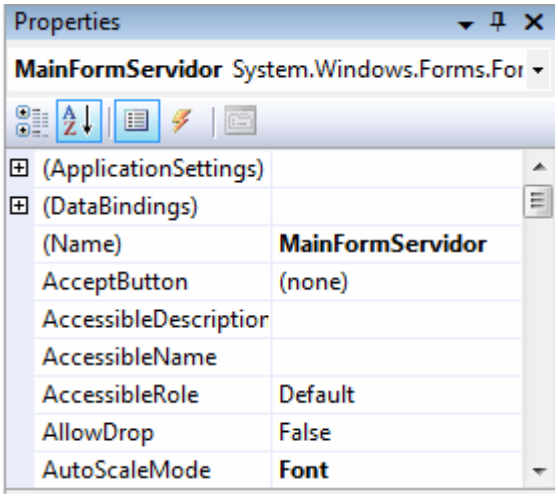

Figura 41: Propiedad nombre del objeto formulario.

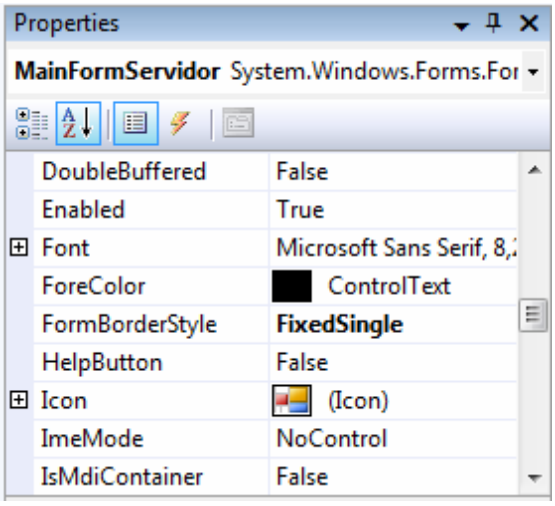

Figura 42: Propiedad tipo de borde del formulario.

3. Cree una nueva clase en su proyecto y nómbrela como *ServidorTcpAeropuerto*, como se indica en la figura 43.

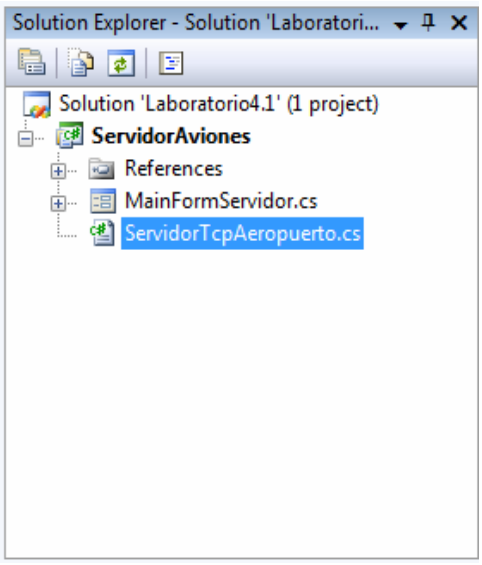

Figura 43: Nueva clase *ServidorTcpAeropuerto* en el proyecto.

4. Utilizando el código de los ejemplos 1 a 4 de esta sección, modifique la clase *ServidorTcpAeropuerto* para que implemente un servidor *TCP* básico con subprocesamiento múltiple. El ejemplo 6 muestra la codificación de toda la clase.

Ejemplo 6

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net.Sockets;
using System.Threading;
using System.Net;

namespace ServidorAviones
{
    class ServidorTcpAeropuerto
    {
        private TcpListener tcpListener;
        private Thread listenThread;

        public ServidorTcpAeropuerto()
        {
          this.tcpListener = new TcpListener(IPAddress.Any, 30000);
          this.listenThread = new Thread(new
ThreadStart(ListenForClients));
          this.listenThread.Start();
        }
        private void ListenForClients()
        {
            this.tcpListener.Start();

            while (true)
            {
                //blocks until a client has connected to the server
                TcpClient client = this.tcpListener.AcceptTcpClient();

                //create a thread to handle communication
                //with connected client
                Thread clientThread = new Thread(new
ParameterizedThreadStart(HandleClientComm));
                clientThread.Start(client);
            }
        }
        private void HandleClientComm(object client)
        {
            TcpClient tcpClient = (TcpClient)client;
            NetworkStream clientStream = tcpClient.GetStream();
            ASCIIEncoding encoder = new ASCIIEncoding();
            byte[] buffer = encoder.GetBytes("Connected");

            clientStream.Write(buffer, 0, buffer.Length);
            clientStream.Flush();

            byte[] message = new byte[4096];
            int bytesRead;

            while (true)
            {
```

```
                bytesRead = 0;

                try
                {
                    //blocks until a client sends a message
                    bytesRead = clientStream.Read(message, 0, 4096);
                }
                catch
                {
                    //a socket error has occured
                    break;
                }

                if (bytesRead == 0)
                {
                    //the client has disconnected from the server
                    break;
                }

                //message has successfully been received
                encoder = new ASCIIEncoding();

System.Diagnostics.Debug.WriteLine(encoder.GetString(message, 0,
bytesRead));
            }
            tcpClient.Close();
        }
    }
}
```

5. Agregue un objeto de tipo *ServidorTcpAeropuerto* al formulario que fue creado en el paso 1 de este laboratorio, o agregue la funcionalidad del servidor como parte del código del formulario. Para cuestiones de este laboratorio, se utiliza la segunda opción. La figura 44 muestra el diseño de la interfaz para este servidor.

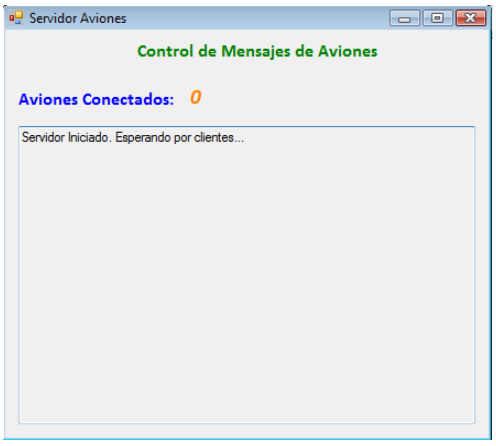

Figura 44: Servidor *TCP* para atención de peticiones de aviones.

6. El ejemplo 7 muestra la implementación completa del formulario junto con el servidor *TCP*.

Ejemplo 7

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Net.Sockets;
using System.Threading;
using System.Net;

namespace ServidorAviones
{
    public partial class MainFormServidor : Form
    {
        private TcpListener tcpListener;
        private Thread listenThread;
        private String lastMessage;
        int clientesConectados;

        public MainFormServidor()
        {
            InitializeComponent();
        }

        private void MainFormServidor_Load(object sender, EventArgs e)
        {
            MensajesRecibidos.Text = "Servidor Iniciado. Esperando por
clientes...\n";
            this.tcpListener = new TcpListener(IPAddress.Any, 30000);
            this.listenThread = new Thread(new
ThreadStart(ListenForClients));
            this.listenThread.Start();
        }

        private void ListenForClients()
        {
            this.tcpListener.Start();

            while (true)
            {
                //blocks until a client has connected to the server
                TcpClient client = this.tcpListener.AcceptTcpClient();

                //create a thread to handle communication
                //with connected client
```

```
                clientesConectados++;
                Conectados.Text = clientesConectados.ToString();
                Thread clientThread = new Thread(new
ParameterizedThreadStart(HandleClientComm));
                clientThread.Start(client);
            }
        }

        private void HandleClientComm(object client)
        {
            TcpClient tcpClient = (TcpClient)client;
            NetworkStream clientStream = tcpClient.GetStream();
            ASCIIEncoding encoder = new ASCIIEncoding();
            byte[] buffer = encoder.GetBytes("Connected");

            clientStream.Write(buffer, 0, buffer.Length);
            clientStream.Flush();

            byte[] message = new byte[4096];
            int bytesRead;

            while (true)
            {
                bytesRead = 0;

                try
                {
                    //blocks until a client sends a message
                    bytesRead = clientStream.Read(message, 0, 4096);
                }
                catch
                {
                    //a socket error has occured
                    break;
                }

                if (bytesRead == 0)
                {
                    //the client has disconnected from the server
                    clientesConectados--;
                    Conectados.Text = clientesConectados.ToString();
                    break;
                }

                //message has successfully been received
                encoder = new ASCIIEncoding();

             System.Diagnostics.Debug.WriteLine(encoder.GetString(message,
            0, bytesRead));
                lastMessage = encoder.GetString(message, 0, bytesRead);
                MensajesRecibidos.Text = "\n" + lastMessage;
            }
            tcpClient.Close();
            clientesConectados--;
            Conectados.Text = clientesConectados.ToString();
        }
    }
```

7. El siguiente paso es agregar la funcionalidad de comunicación en red a nuestra clase Avión. Para ello se recomienda realizar un nuevo proyecto en *Visual C#,* copiar la clase y el formulario Avión del laboratorio 3.1 en él y modificarlos.

8. Excluya del formulario Avión todas las funciones relacionadas con bases de datos. Su clase debe quedar similar al ejemplo 8.

Ejemplo 8

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Data.SqlClient;

namespace AvionTCP
{
    public partial class AvionSockets : Form
    {
        private Avion aeronave;
        enum EstadosAvion { Tierra, Despegar, Volar, Aterrizar };

        public AvionSockets()
        {
            InitializeComponent();
            aeronave = new Avion();
        }

        public AvionSockets(String Matricula, String Airline)
        {
            InitializeComponent();
            aeronave = new Avion();
            placaAvion.Text = Matricula;
            estadoAvion.Text = aeronave.Estado;
            aerolineaAvion.Text = Airline;

        }
        private void Despegar_Click(object sender, EventArgs e)
        {
            if (aeronave.cambiarEstado((int)EstadosAvion.Despegar))
            {

            }
            else
```

```
            {
                MessageBox.Show("El cambio de estado no es
coherente. Intente de nuevo", "Cambio Incorrecto",
MessageBoxButtons.OK);
            }
        }

        private void Volar_Click(object sender, EventArgs e)
        {
            if (aeronave.cambiarEstado((int)EstadosAvion.Volar))
            {

            }
            else
            {
                MessageBox.Show("El cambio de estado no es
coherente. Intente de nuevo", "Cambio Incorrecto",
MessageBoxButtons.OK);
            }
        }

        private void Aterrizar_Click(object sender, EventArgs e)
        {
            if  (aeronave.cambiarEstado((int)EstadosAvion.Aterrizar))
            {

            }
            else
            {
                MessageBox.Show("El cambio de estado no es
coherente. Intente de nuevo", "Cambio Incorrecto",
MessageBoxButtons.OK);
            }
        }

        private void Descargar_Click(object sender, EventArgs e)
        {
            if (aeronave.cambiarEstado((int)EstadosAvion.Tierra))
            {

            }
            else
            {
                MessageBox.Show("El cambio de estado no es
coherente. Intente de nuevo", "Cambio Incorrecto",
MessageBoxButtons.OK);
            }
        }

        private void AvionSockets_Load(object sender, EventArgs e)
        {

        }
    }
}
```

9. La figura 45 muestra un ejemplo del formulario que fue creado para este laboratorio con el fin de representar el Avión.

Figura 45: Representación de un avión.

10. Agregue la funcionalidad al evento *click* del botón Conectar. El ejemplo 9 muestra como fue implementado en la solución programada que se encuentra en la plataforma *Web*.

Ejemplo 9

```
private void btnConectar_Click(object sender, EventArgs e)
{
   if (IPServidor.Text == "")
   {
      MessageBox.Show("Por favor ingrese una dirección IP para el
servidor", "Error al conectar", MessageBoxButtons.OK,
MessageBoxIcon.Exclamation);
   }
   else
```

```
    {
        if (clienteRed.Connected)
        {
            try
            {
                serverEndPoint = new
IPEndPoint(IPAddress.Parse(IPServidor.Text), 30000);
                clienteRed.Connect(serverEndPoint);

                clientStream = clienteRed.GetStream();

                encoder = new ASCIIEncoding();
                byte[] buffer = encoder.GetBytes("Hello Server!");

                clientStream.Write(buffer, 0, buffer.Length);
                clientStream.Flush();
                lblConectado.ForeColor = Color.Green;
                lblConectado.Text = "Conectado";
            }
            catch (Exception Ex)
            {
                MessageBox.Show(Ex.ToString());
            }
        }
        else {
            try
            {
                clienteRed.Close();
                lblConectado.ForeColor = Color.Red;
                lblConectado.Text = "Desconectado";
            }
            catch (Exception Exc) {
                MessageBox.Show(Exc.ToString());
            }
        }
    }
}
```

11. Preste atención al ejemplo 9 y observe cómo se utilizan los atributos de la clase *TcpClient* tanto para obtener la cadena de escritura al *socket* como para determinar si el cliente se encuentra conectado a la aplicación servidor. Este evento programa tanto las funciones de conectar como de desconectar para el avión *TCP*.

12. A continuación, programe cada uno de los eventos de los botones de la interfaz del avión. El ejemplo 10 muestra cómo se realizó dicha programación para el evento *click* del botón despegar.

Ejemplo 10

```
private void Despegar_Click(object sender, EventArgs e)
{
   if (clienteRed.Connected)
   {
      if (Aeronave.cambiarEstado((int)EstadosAvion.Despegar))
      {
         String mensaje = placaAvion.Text + "." +
aerolineaAvion.Text + "." + Aeronave.Estado;
         clientStream = clienteRed.GetStream();
         encoder = new ASCIIEncoding();
         byte[] buffer = encoder.GetBytes(mensaje);
         clientStream.Write(buffer, 0, buffer.Length);
         clientStream.Flush();
      }
      else
      {
         MessageBox.Show("El cambio de estado no es coherente.
Intente de nuevo", "Avión TCP", MessageBoxButtons.OK);
      }
   }
   else
   {
      MessageBox.Show("Debe establecer una conexión antes de
ejecutar cualquier acción", "Avión TCP", MessageBoxButtons.OK,
MessageBoxIcon.Exclamation);
   }
}
```

13. A continuación, pruebe la aplicación y conéctela a través de la *IP* 127.0.0.1. El servidor deberá informar que ha recibido un nuevo cliente. Cree otra aplicación Avión y conéctela también. El servidor deberá manejar a todos los clientes sin problemas. Cuando los mensajes arriben, deberán ser visibles en el espacio compartido de pantalla del servidor.

14. Las figuras 46 y 47 muestran un ejemplo del funcionamiento de la aplicación servidor y dos clientes simultáneos; uno conectado y otro que ya completó su ejecución y está desconectado.

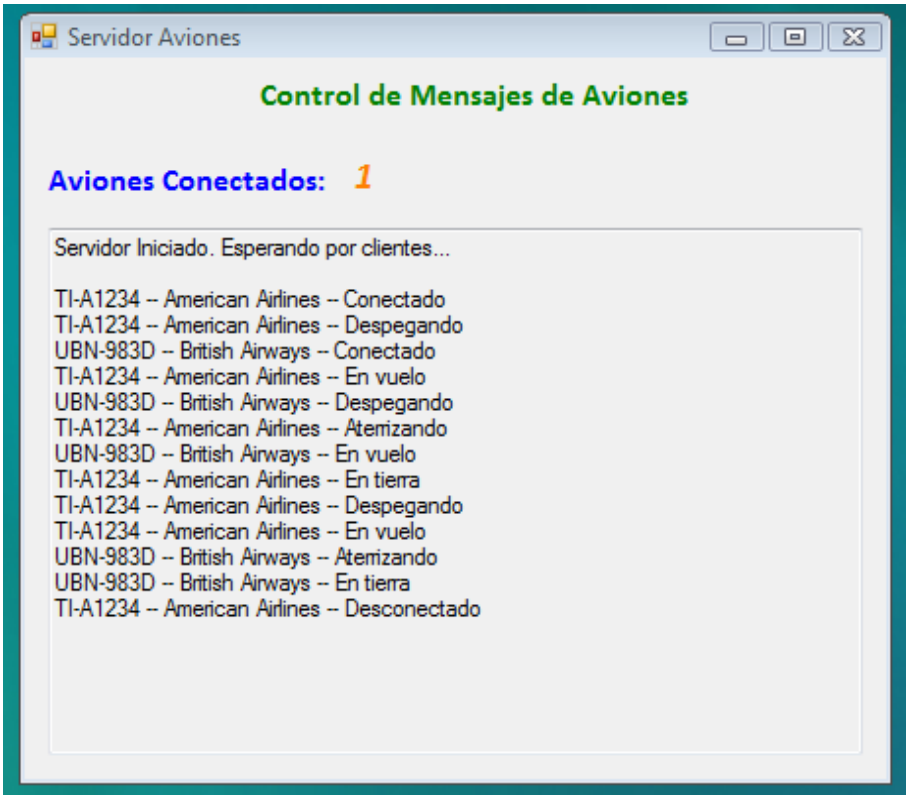

Figura 46: Ventana de funcionamiento del servidor *TCP*.

Figura 47: Dos clientes de tipo *AvionTCP* en ejecución.

15. La solución a este problema que se encuentra en la plataforma *Web* tiene algunas mejoras funcionales y correcciones a "*bugs*". Descargue la última versión antes de ejecutar este laboratorio.

Laboratorio 4.2

El código Morse es uno de los sistemas de codificación de mensajes más famosos. Fue desarrollado por Samuel Morse en el año 1832 para facilitar la transmisión de mensajes a través del telégrafo. Este código asigna una serie de puntos y comas a cada letra del alfabeto, así como a cada número y a algunos caracteres especiales, como la coma, el punto y el punto y seguido. El patrón de este sistema puede ser utilizado con otros elementos tales como una luz o sonido, donde un espacio entre palabras se indica por un corto período de tiempo donde no se transmite sonido. La figura 48 muestra la versión internacional del código Morse.

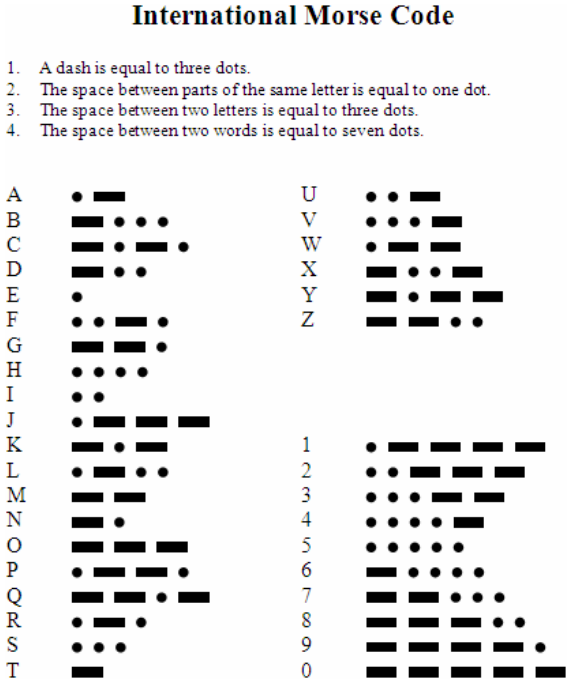

Figura 48: Código Morse Internacional.
Fuente:< http://nancydrewology.files.wordpress.com>.

A. Elabore una aplicación cliente que realice lo siguiente:

1. Leer una frase en español desde un *TextBox* y convertirla a código Morse.
    - Para este proceso, use un espacio en blanco entre cada letra y tres espacios en blanco entre cada palabra cuando utilice código Morse.
2. Enviar esta frase codificada a un servidor de red multiproceso por el puerto *TCP* 50505.
3. Recibir una frase desde el servidor codificada en código Morse, convertirla a español y presentarla al usuario en un *Textbox*.

B. Elabore una aplicación servidor que realice lo siguiente:

1. Intercomunicar los clientes Morse a través del protocolo *TCP* y utilizando subprocesamiento múltiple.

2. Guardar una bitácora de todos los mensajes recibidos mientras el servidor se encuentre operando. Debe mostrar estos mensajes en pantalla tanto en código Morse como en español.

3. Para esta solución, puede utilizar el ejemplo del "*ChatNetwork*" que se brinda como referencia adicional en esta sección de la guía, o el que se mostró en los ejemplos 1 a 3 de esta sección de la guía. Modifíquelos según sus necesidades.

Este ejercicio será realizado en la cuarta tutoría del curso. Recuerde que puede encontrar una solución programada a este ejercicio en la plataforma virtual, en caso de que no pueda asistir. No olvide documentar los aprendizajes de los ejercicios en sus reportes de laboratorio.

Glosario

Encabezado: Un encabezado o *heade,* es la sección inicial de un paquete de datos. Esta varía según el protocolo que está siendo transmitido y contiene información que determina la función del paquete. Un ejemplo es el encabezado del protocolo de correo, que contiene la información del destinatario, del emisor, así como el título del mensaje. El encabezado del protocolo *IP* indica la dirección destino, la dirección fuente la información, así como el protocolo de capa superior que está siendo encapsulado dentro del paquete *IP.*

Protocolo enrutado: Cualquier protocolo que contiene información suficiente en su encabezado para permitir que un enrutador envíe el paquete de datos hacia otro a través de la red.

Protocolo de enrutamiento: Los protocolos de enrutamiento son aquellos que utilizan los *routers* para intercambiar información de rutas y topología de la red, con el fin de establecer caminos por los cuales enviarán la información que reciban. De esta forma, permiten encaminar los paquetes de los protocolos enrutados.

Servidor *Proxy:* **"**Un servidor *proxy* es un equipo que actúa de intermediario entre un explorador *Web* (como Internet Explorer) e Internet. Los servidores *proxy* ayudan a mejorar el rendimiento en Internet ya que almacenan una copia de las páginas *Web* más utilizadas. Cuando un explorador solicita una página *Web* almacenada en la colección (su caché) del servidor *proxy,* el servidor *proxy* la proporciona, lo que resulta más rápido que consultar la *Web*. También ayudan a mejorar la seguridad, ya que filtran algunos contenidos *Web* y *software* malintencionado. Los servidores *proxy* se utilizan a menudo en redes de organizaciones y compañías. Normalmente, las personas que se conectan a Internet desde casa no usan un servidor *proxy*." (Microsoft, Inc.)

Fuentes consultadas

Meza, S. (2005). *Sistemas Operativos Multiusuario.* Ciudad de México, México, DF, México.

Tanenbaum, A. S. (2001). *Systemas Operativos Modernos.* New Jersey: Prentice Hall.

Microsoft, Inc. (s.f.). *¿Qué es un servidor proxy?* Recuperado el 19 de 06 de 2009, de Windows Help: <http://windowshelp.microsoft.com/Windows/es-ES/help/7ced6d16-fb98-475f-aaca-87f820b3a9873082.mspx>.

Microsoft, Inc. (s.f.). *Semaphore (Clase): MSDN*. Recuperado el 18 de Mayo de 2009, de MSDN: <http://msdn.microsoft.com/es-es/library/system.threading.semaphore(VS.80).aspx>.

Soto, L. (s.f.). *Instituto Tecnológico de Tijuana.* Recuperado el 14 de Mayo de 2009, de <http://www.mitecnologico.com/Main/ConceptoDeHilo>.

Wikipedia. (s.f.). *Wikipedia.* Recuperado el 14 de Mayo de 2009, de <http://es.wikipedia.org/wiki/Bloqueo_mutuo>.