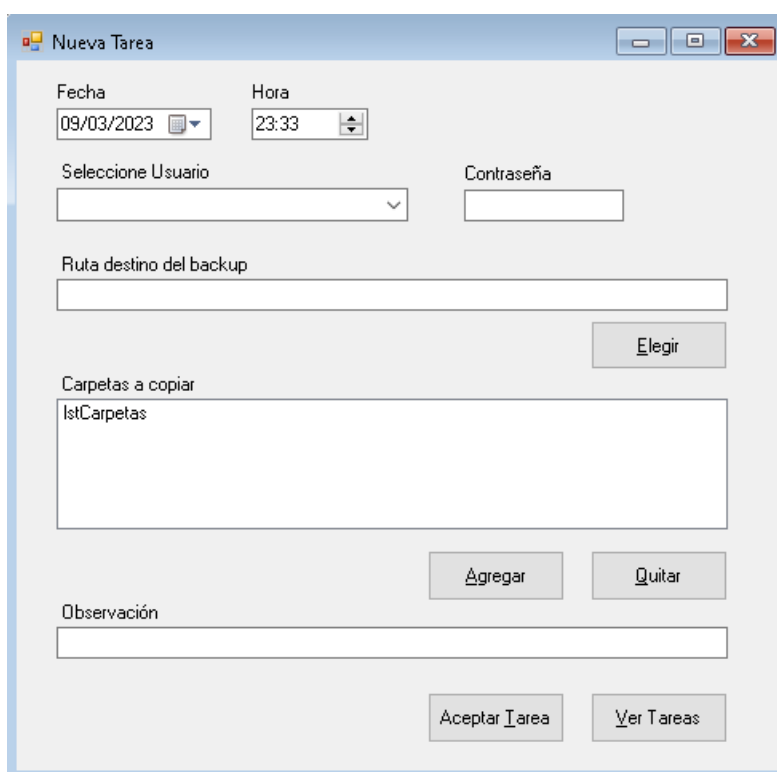


Situación profesional 2: Backup Programado

El administrador de una empresa del rubro automotriz le ha solicitado a usted, el desarrollo de una aplicación para que los empleados puedan almacenar tareas programadas con fecha y hora para el respaldo automático de la información generada en los distintos sectores de la empresa. El procedimiento para almacenar una tarea es la siguiente: seleccionar la fecha y hora de ejecución del respaldo, seleccionar el usuario responsable, ingresar la contraseña del mismo, la ruta para realizar la copia, las rutas de las carpetas a copiar y una observación sobre la copia.

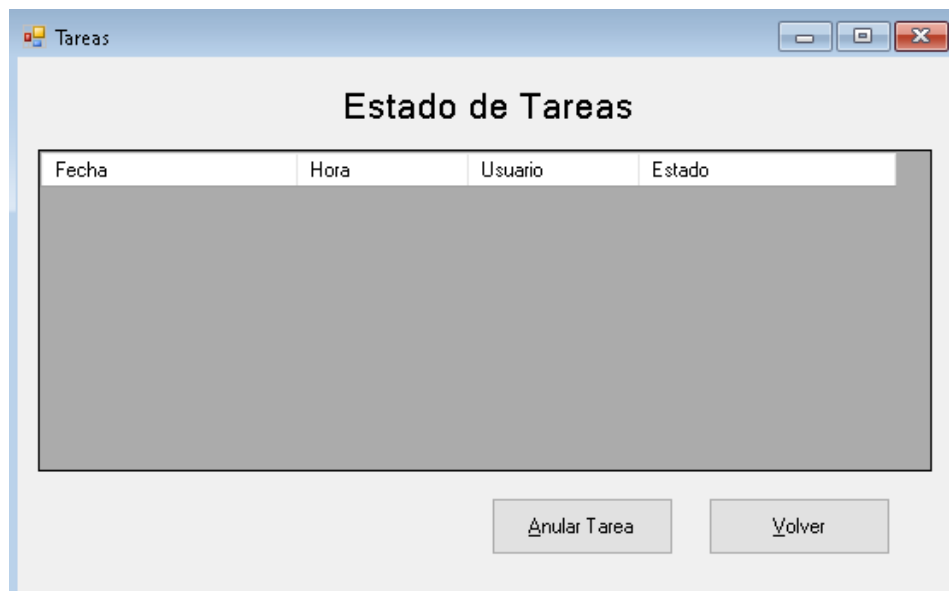


The image shows a Windows-style application window titled "Nueva Tarea". It contains several input fields and buttons. At the top, there are "Fecha" (Date) and "Hora" (Time) fields. Below them are "Seleccione Usuario" (Select User) and "Contraseña" (Password) fields. Then, there is a "Ruta destino del backup" (Backup destination path) field with an "Elegir" (Choose) button. Below that is a "Carpetas a copiar" (Folders to copy) list box containing "IstCarpetas". There are "Agregar" (Add) and "Quitar" (Remove) buttons next to the list. At the bottom, there is an "Observación" (Observation) field and two buttons: "Aceptar Tarea" (Accept Task) and "Ver Tareas" (View Tasks).

Antes de almacenar una tarea se debe controlar que no exista una tarea con la fecha y hora seleccionada, la fecha y la hora no puede ser menor a la fecha y la hora actual, la contraseña del usuario debe ser correcta, la ruta destino tiene que ser correcta y se tiene que haber elegido al menos una carpeta para copia, no permita copiar rutas de carpetas repetidas y la carpeta destino nunca puede ser igual a una carpeta origen, en todos los casos informe el error correspondiente.

También el usuario tiene que poder ver todas las tareas almacenadas y poder anular cualquier tarea que se encuentre pendiente de ejecutar. Los estados de las tareas son: pendiente,

terminada o anulada. De cada tarea se muestra la fecha, la hora, el nombre del usuario responsable y el estado de la tarea.



El proceso temporizado se tiene que ejecutar cada 30 segundos. Este proceso es el encargado de ejecutar todas las tareas pendientes al cumplirse la fecha y la hora programada. Para llevar a cabo las tareas mencionadas anteriormente, se cuenta con una base de datos que tiene tres tablas: ·

La tabla usuarios almacena un número para identificar al usuario, el nombre del usuario y su contraseña, la clave principal de la tabla es el número de usuario. ·

La tabla tareas almacena la fecha y hora de ejecución de la tarea, el número de usuario que generó la tarea, la ruta destino dónde se realizará la copia, una observación hecha por el usuario y el estado de la tarea (0=pendiente, 1=terminada o 2=anulada), la clave principal de la tabla está formada por las columnas fecha y hora. ·

La tabla carpetas almacena la fecha, la hora, el orden de la copia y la ruta de la carpeta a copiar, la clave principal de la tabla está formada por las columnas fecha, hora y orden.

Usuarios:

Usuarios	
Nombre del campo	Tipo de datos
usuario	Número
nombre	Texto corto
palabra	Texto corto

Tareas:

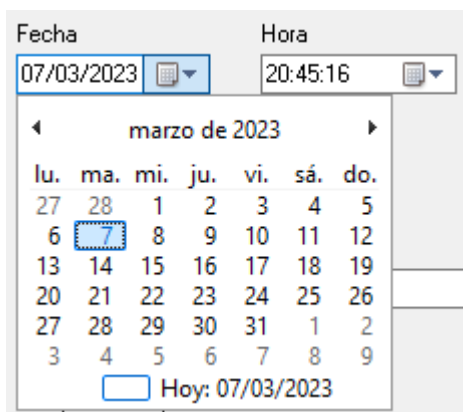
Tareas	
Nombre del campo	Tipo de datos
fecha	Fecha/Hora
hora	Texto corto
usuario	Número
rutadestino	Texto corto
observacion	Texto corto
estado	Número

Carpetas:

Carpetas		Crear macros
Nombre del campo	Tipo de datos	
fecha	Fecha/Hora	
hora	Texto corto	
orden	Número	
rutaorigen	Texto corto	

SP2/H1: Control DateTimePicker

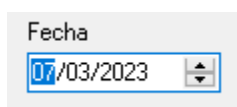
El control DateTimePicker se usa para permitir al usuario seleccionar una fecha y hora, y mostrar esa fecha y hora en el formato específico. El control DateTimePicker facilita el trabajo con fechas y horas, ya que controla las validaciones de datos automáticamente.



A la izquierda de la imagen: DateTimePicker configurado con formato de fecha corta, al activarlo despliega un calendario para seleccionar una fecha de manera más cómoda, la ventana del calendario permite cambiar también el mes y el año.

A la derecha se presenta otro control DateTimePicker, pero en este caso está configurado para mostrar la hora en formato “hh:mm:ss”.

Otra posibilidad de uso frecuente es reemplazar la ventana que muestra el calendario por dos botones “up-down”, para subir o bajar los valores del día, mes o año según qué parte se seleccione. La propiedad “ShowUpDown” en verdadero activa este modo y desactiva el calendario:



Este formato puede resultar más cómodo cuando tenemos disponible un rango de fechas acotado. Las propiedades “MinDate” y “MaxDate” permiten establecer los valores mínimos y máximos entre los que se podrán elegir las fechas.

La propiedad “Value” devuelve el valor seleccionado en un tipo DateTime, es decir que contiene siempre la fecha y la hora. El valor de la hora será la hora del sistema salvo que el

control DateTimePicker esté configurado para mostrar la hora en cuyo caso el valor de la hora será el configurado por el usuario en el control.

Los valores del control pictureBox se pueden mostrar en cuatro formatos, que se establecen mediante la propiedad "Format": "Long" (fecha larga), "Short" (fecha corta), "Time" (hora) o "Custom" (personalizado). El valor predeterminado de la propiedad "Format" es "Long".

Si desea que aparezca el control DateTimePicker para seleccionar o editar horas en lugar de fechas, establezca la propiedad "ShowUpDown" en true y la propiedad "Format" en Time.

Si la propiedad "Format" está establecida "DateTimePickerFormat.Custom", puede crear su propio estilo de formato estableciendo la propiedad "CustomFormat" y creando una cadena de formato personalizada.

La cadena de formato personalizado puede ser una combinación de caracteres de campo personalizados y otros caracteres literales. Por ejemplo, puede mostrar la fecha como "Junio 1, 2012 Viernes" estableciendo la propiedad "CustomFormat" en "MMMM dd, yyyy - dddd".

```
DateTimePicker.Format = DateTimePickerFormat.Custom;  
DateTimePicker.CustomFormat = "MMMM dd, yyyy dddd";
```

En el caso de necesitar establecer el valor de la fecha y el valor de la hora en el mismo control DateTimePicker podemos asignar este formato:

```
DateTimePicker.Format = DateTimePickerFormat.Custom;  
DateTimePicker.CustomFormat = "dd/MM/yyyy hh:mm:ss";
```

Puede revisar la lista completa de propiedades y métodos del control "DateTimePicker" es este enlace:

<https://learn.microsoft.com/es-es/dotnet/api/system.windows.forms.datetimepicker?view=windowsdesktop-7.0>

1. Indique la opción correcta:

El control DateTimePicker permite mostrar los valores de fecha y hora solamente con formatos fijos y preestablecidos.

- Verdadero
- Falso X

2. Indique la opción correcta:

El nombre de la propiedad que permite obtener el valor seleccionado en el control DateTimePicker es:

- Text
- Value X
- Date
- Ninguno de los anteriores

3. Indique la opción correcta:

En un control DateTimePicker es posible fijar las fechas mínimas y máximas que admite como válidas.

- Verdadero X
- Falso

4. Indique la opción correcta:

Para que el control DateTimePicker muestre la ventana con el calendario desde el botón incluido a su derecha se debe ajustar el valor de la propiedad:

- ShowUpDown X
- ShowCheckBox
- ShowCalendar

5. Indique la opción correcta:

El control DateTimePicker valida automáticamente si una fecha ingresada manualmente es válida o no.

- Verdadero X
- Falso

SP2/H2: Control Timer

En esta Situación profesional se plantea la necesidad de que la aplicación lleve un control del tiempo en el que van a transcurrir ciertas tareas. Para ello utilizaremos un control muy útil que es el Temporizador (Timer).

El control Timer responde al paso del tiempo. Es independiente del usuario de la aplicación, y se puede programar para que ejecute acciones a intervalos periódicos de tiempo. Un uso típico es comprobar la hora del sistema para ver si es el momento de ejecutar alguna tarea, como es lo que deseamos realizar para la resolución de nuestra situación profesional. Los cronómetros también son útiles para otro tipo de procesamiento en segundo plano.

Los controles Timer deben estar asociados a un formulario, por lo tanto, para crear una aplicación de cronómetro, debe crear al menos un formulario, aunque no es obligatorio que sea visible, si no se necesita para otro fin.

Propiedades Principales:

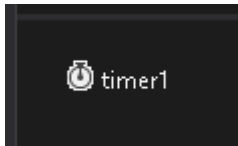
Interval: Número de milisegundos que deben transcurrir para ejecutar el evento Tick. El evento Tick es periódico. La propiedad Interval no determina la duración sino la frecuencia. La duración del intervalo depende de la precisión que se desea. Como existe cierta posibilidad de error, se recomienda que la precisión del intervalo sea el doble de la deseada. Se recomienda no fijar intervalos demasiado cortos, a menos que sea realmente necesario.

Enabled (verdadero o falso). Determina si el control Timer realiza el conteo de los milisegundos establecidos en la propiedad Interval. Cuando su valor es True, el cronómetro empieza a funcionar, cuando su valor es False el cronómetro se detiene. Esta propiedad se puede establecer en tiempo de diseño o en tiempo de ejecución.

Eventos del control Timer:

El principal evento se denomina **Tick**: este evento se genera en forma automática cada vez que el cronómetro alcanza el valor establecido en la propiedad Interval, y se repetirá en forma periódica mientras la propiedad Enabled del timer sea True. El control Timer es un control invisible en la interfaz de la aplicación y, por lo tanto, no se mostrará al usuario, solamente se visualiza en tiempo de diseño.

Ejemplo demostración: En el formulario agregamos un control Timer:



En el evento Load del formulario establecemos el valor de la propiedad “Interval” en 1000, serían 1000 milisegundos = 1 segundo, y habilitamos la ejecución del timer colocando “true” en la propiedad “Enabled”.

```
private void Form1_Load(object sender, EventArgs e)
{
    timer1.Interval = 1000;
    timer1.Enabled = true;
}

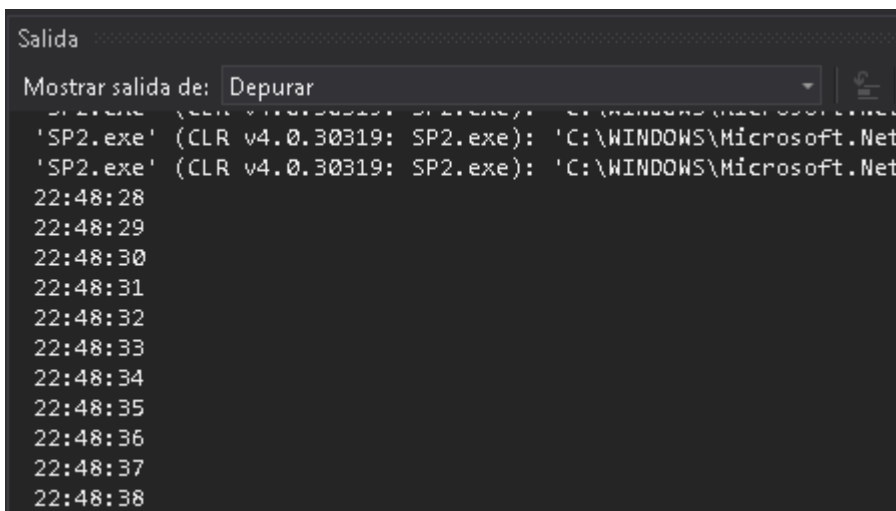
private void timer1_Tick(object sender, EventArgs e)
{
    // el evento se ejecuta una vez por segundo
    Trace.WriteLine(DateTime.Now.ToLongTimeString());
}
```

Haciendo doble click sobre el icono del timer en el formulario generamos el evento “Tick”, y en él escribimos una llamada al método WriteLine del objeto Trace, el valor que se escribirá en la consola de salida de Visual Studio será el valor de la hora del sistema con formato “hh:mm:ss”, eso ocurrirá una vez cada un segundo porque así está determinado por la propiedad “Interval”.

Tenga en cuenta que para poder usar la clase Trace se debe agregar al código del formulario:

```
using System.Diagnostics;
```

Al ejecutar el formulario podemos ver la consola de salida con los valores de la hora cada un segundo, esto es porque el evento “Tick” del timer se ejecuta automáticamente cada 1 segundo.



En un caso real el evento “Tick” es el que lleva el control del proceso que se desea ejecutar en forma periódica de acuerdo al valor de la propiedad “Interval”. En ese evento se deberá colocar toda la lógica del proceso que necesitamos ejecutar periódicamente.

En nuestra situación profesional deberemos agregar el control TIMER para poder realizar las tareas necesarias y controlar la ejecución de las copias de las carpetas de backup en la fecha y hora registradas en la base de datos.

Podrá consultar toda la documentación del control Timer en este enlace:

<https://learn.microsoft.com/en-us/dotnet/api/system.timers.timer?view=netframework-4.8.1>

1. Ordene relaciones. Relacione los conceptos de la columna izquierda con sus características correspondientes en la columna derecha:

Interval	Tiene lugar cuando ha transcurrido el intervalo de tiempo especificado. (Tick)
Enabled	Habilita la generación de eventos con el intervalo Establecido (Enabled)
Tick	Frecuencia de los eventos en milisegundos. (Interval)

2. Indique la opción correcta

¿Cuál es el evento que nos permite agregar código para manejar la acción de un Timer?:

Evento Tick X
Enabled
Changed
Load

3. Indique la opción correcta

¿Cuál es la propiedad que nos permite habilitar la cuenta del tiempo transcurrido en un control Timer?:

Interval
Time
Enabled X
Tick

4. Indique la opción correcta

Cuando se cumple el intervalo de un control Timer, se genera un evento Tick.

Verdadero X
Falso

5. Indique la opción correcta

El control Timer puede mostrarse en la interfaz o no según el valor de la propiedad Visible.

Verdadero

Falso X

SP2/H3: Transacciones en ADO .NET

Las transacciones están relacionadas a las operaciones sobre la base de datos y nos permiten tratar varias operaciones como una sola unidad, como si fueran una sola operación atómica, este tratamiento tiene por objetivo mantener la integridad de los datos entre las distintas tablas que participan de esas operaciones individuales. Si al ejecutar algunos de los pasos que componen la transacción falla entonces la transacción se cancela y todos los cambios realizados hasta ese momento se revierten dejando todo en el estado inicial, si todos los pasos se completan exitosamente entonces la transacción finaliza confirmando todos los cambios. Este mecanismo tiene entonces tres estados: “Begin” para dar inicio a la transacción, “Commit” para confirmar todos los cambios realizados y “Rollback” para cancelar y revertir todos los cambios realizados.

Una transacción se caracteriza por cumplir las llamadas propiedades **ACID**, que identifican los requisitos para que una transacción se realice. Estas propiedades son: atomicidad (**A**tomicity), coherencia (**C**onsistency), aislamiento (**I**solation) y permanencia (**D**urability).

Desde ADO .NET podemos trabajar con transacciones a partir del objeto “Connection”, este objeto posee los métodos para controlar el inicio, confirmación o cancelación de las transacciones:

- BeginTransaction
- Commit
- Rollback

Con el objeto “Transaction” creado al ejecutar “BeginTransaction” se controlará todo el proceso incluido en la transacción, todos los comandos ejecutados deben llevar asociado el objeto transacción creado al comienzo para poder determinar si algo falla o se completa exitosamente.

Ejemplo: supongamos que debemos hacer un proceso que trabaja con 2 tablas: “Movimientos” y “Cuentas”. La tabla “Movimientos” registra las operaciones de depósito que se realizan en cada cuenta, la tabla “Cuentas” registra el saldo actual de cada cuenta. Cuando se quiere procesar una operación de depósito se debe agregar un registro nuevo a la tabla “Movimientos” y luego editar la tabla de “Cuentas” para ajustar el saldo de acuerdo al importe del depósito,

son dos operaciones independientes, cada una trabaja con una tabla diferente. Si no usamos transacciones para agrupar ambas operaciones podría ocurrir que luego de procesar correctamente el movimiento ocurra un error en el proceso de la cuenta, lo que nos dejaría cargado un movimiento cuyo importe no está reflejado en el saldo de la cuenta, con una transacción eso no puede ocurrir porque ante cualquier error todos los cambios se revierten y se vuelve al estado inicial.

Código principal que hace uso de una transacción: para simplificar el ejemplo asumimos que la conexión con la base de datos ya está creada previamente, es el objeto “CNN”, la conexión debe estar abierta antes de iniciar la transacción.

```
public void AddMovimiento()
{
    OleDbTransaction Transaccion = null;
    try
    {
        // iniciar la transaccion (CNN es el objeto OleDbConnection)
        Transaccion = CNN.BeginTransaction();
        // agregar un registro nuevo en Movimientos
        InsertMovimiento(Transaccion);
        // actualizar el registro de la cuenta con el nuevo saldo
        UpdateCuenta(Transaccion);
        // confirmar todos los cambios realizados
        Transaccion.Commit();
    }
    catch (Exception ex)
    {
        // revertir los cambios de la transacción
        Transaccion.Rollback();
    }
}
```

Como se observa en el código anterior, el método “BeginTransaction” crea el objeto y da inicio al proceso de transacción, luego se realizan las operaciones que se necesiten completar como si fuera una única operación, en este ejemplo hay dos operaciones, la primera inserta un registro nuevo en la tabla de Movimientos y la segunda actualiza el saldo en la tabla de Cuentas, si ambas operaciones se completan sin error se ejecuta el método “Commit” que se encarga de confirmar sobre la base de datos todos los cambios realizados en los pasos anteriores. Si se produce algún error el bloque catch() captura la excepción y se ejecuta el método “Rollback” de la transacción, revirtiendo todos los cambios previos y dejando la base de datos en el estado inicial que tenía antes de iniciar la transacción.

Un detalle importante es que cada una de las operaciones que forman parte de la transacción deben asignar el objeto Transacción creado al objeto “Command” como se puede ver en las dos funciones siguientes:

```
public void InsertMovimiento(OleDbTransaction transaccion)
{
    // asignar la transacción al comando de Movimientos
    CmdMov.Transaction = transaccion;
    // agregar un nuevo registro a la tabla de Movimientos
    DataRow dr = DS.Tables["Movimientos"].NewRow();
    dr["Fecha"] = fecha;
    dr["Importe"] = importe;
    dr["Cuenta"] = cuenta;
    DS.Tables["Movimientos"].Rows.Add(dr);
    OleDbCommand cb = new OleDbCommandBuilder(DAMov);
    DAMov.Update(DS, "Movimientos");
}
```

Este método se encarga de agregar un nuevo registro en la tabla de Movimientos, el proceso ya se conoce y no necesita mayores comentarios, la única salvedad es esta línea:

```
// asignar la transacción al comando de Movimientos
CmdMov.Transaction = transaccion;
```

El objeto “Command” tiene la propiedad “Transaction” y debe ser asignada con el objeto “Transaction” creado inicialmente con el método “BeginTransaction” para que la transacción pueda conocer todas las operaciones que deben ser controladas y finalmente confirmadas o canceladas.

Algo similar resulta en el método para actualizar el saldo de la cuenta:

```
public void UpdateCuenta(OleDbTransaction transaccion)
{
    // asignar la transacción al comando de Cuentas
    CmdCta.Transaction = transaccion;
    // actualizar el registro de la cuenta con el nuevo saldo
    DataRow dr = DS.Tables["Cuentas"].Rows.Find(cuenta);
    dr.BeginEdit();
    dr["Saldo"] = dr["Saldo"] + importe;
    dr.EndEdit();
    OleDbCommand cb = new OleDbCommandBuilder(DACta);
    DACta.Update(DS, "Cuentas");
}
```

En este ejemplo se asume que todos los objetos DataSet, Command y DataAdapter, como así también los valores de los campos que se usan para asignar en cada tabla son todos accesibles desde cada uno de los métodos programados, por ejemplo, declarados como miembros privados de la clase. En casos reales, se recomienda que esos objetos y variables sean parámetros de cada uno de los métodos.

Puede consultar este tema en el enlace oficial de Microsoft:

<https://learn.microsoft.com/es-es/dotnet/framework/data/adonet/local-transactions>

1. Indique la opción correcta.

Una transacción permite procesar varias operaciones como si fueran una sola unidad, si alguna operación falla se descartan todos los cambios realizados.

Verdadero ☒ X

Falso

2. Indique la opción correcta.

El objetivo de las transacciones es:

Mantener la integridad de los datos ☒ X

Ejecutar más rápido las operaciones

Delegar los controles a la base de datos

3. Indique la opción correcta.

Indique cuál es la secuencia correcta en la ejecución de las etapas de una transacción

Begin, Rollback, Commit

Commit, Rollback, Begin

Begin, Commit, Rollback ☒ X

Rollback, Begin, Commit

4. Indique la opción correcta.

En ADO .NET el manejo de las transacciones se genera de la conexión con la base de datos.

Verdadero ☒ X

Falso

5. Indique la opción correcta.

Una vez confirmada la transacción ya no es posible revertir los cambios en forma automática.

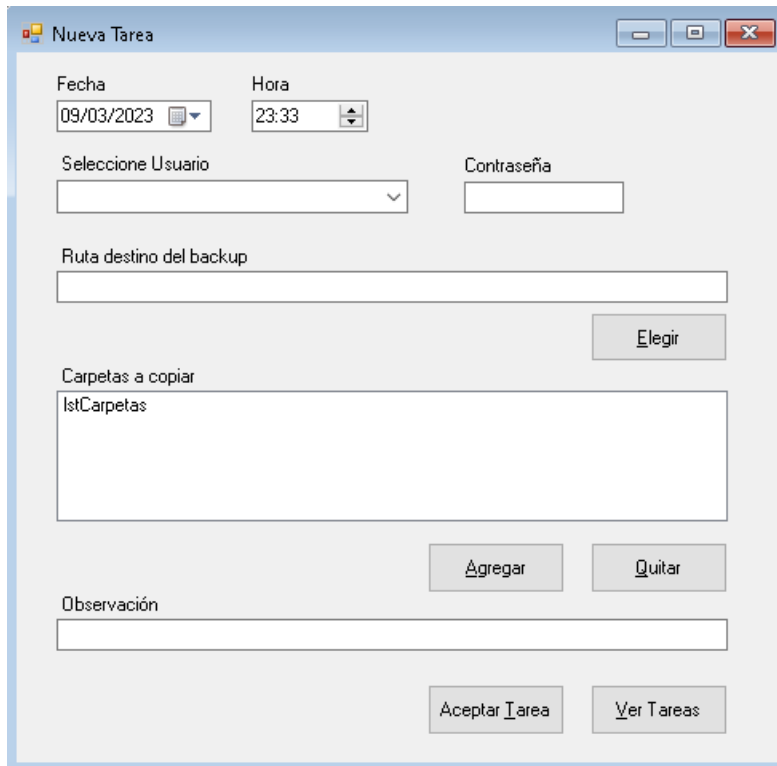
Verdadero ☒ X

Falso

SP2/Ejercicio resuelto

Comenzaremos la resolución de esta situación profesional creando el proyecto y diseñando la interfaz gráfica de cada formulario:

Form1:



Form1: Nueva Tarea

Fecha: 09/03/2023 Hora: 23:33

Seleccione Usuario: [dropdown] Contraseña: [input]

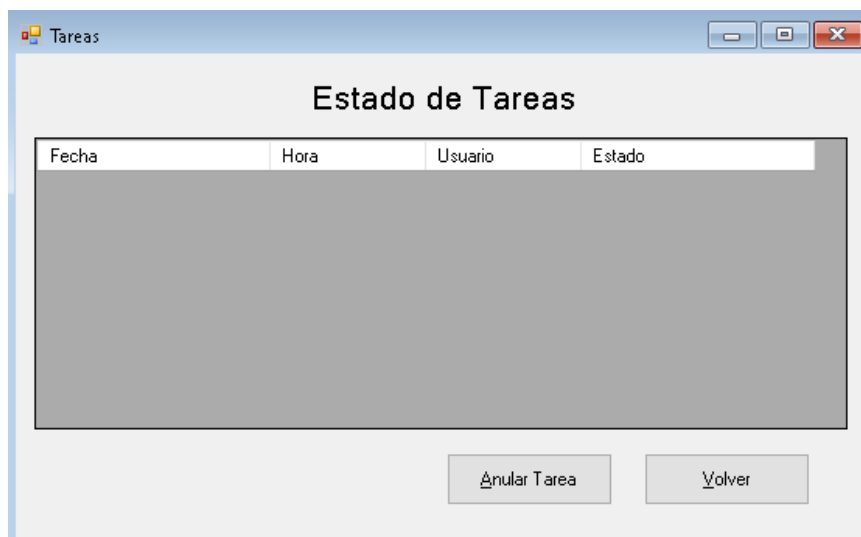
Ruta destino del backup: [input] [Elegir]

Carpeta a copiar: [input] [Agregar] [Quitar]

Observación: [input]

[Aceptar Tarea] [Ver Tareas]

Form2:



Form2: Tareas

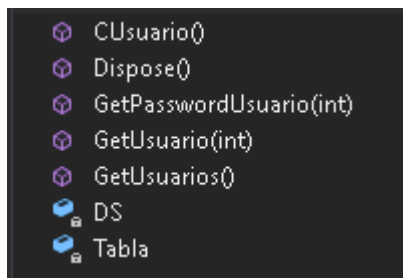
Estado de Tareas

Fecha	Hora	Usuario	Estado
-------	------	---------	--------

[Anular Tarea] [Volver]

Seguidamente agregaremos las clases necesarias para manejar las tablas de la base de datos y los procedimientos que debemos realizar en ellas. Tendremos solamente 2 clases: “CUusuario” y “CTarea”, describiremos el contenido y la implementación de cada clase.

Clase “**CUusuario**”, se encargará de acceder a la tabla de Usuarios por medio del constructor de la clase, y tendrá los métodos “Dispose” para liberar los recursos empleados, “GetPasswordUsuario” para recuperar el valor del password de un usuario determinado, “GetUsuario” para obtener el nombre de un usuario y “GetUsuarios” para obtener la tabla completa con todos los usuarios.



Clase “**CTarea**”, la idea es similar a la clase anterior, pero en este caso se controlarán 2 tablas: la tabla de Tareas y la tabla de Carpetas, esto es porque tenemos una relación y dependencia entre las tablas, una tarea deberá tener una o varias carpetas asociadas para hacer las copias de archivos. Los métodos serán, además del constructor y del método Dispose, los métodos “AddTarea” encargada de agregar una nueva tarea a la base de datos, internamente ejecutará otros dos métodos, “InsertTarea” e “InsertRutas” para distribuir los datos en las 2 tablas de la base. Tenemos también el método “GetTareas” que devuelve la tabla de tareas completa, el método “EjecutarTarea” que será el método invocado desde el evento “Tick” del control timer y finalmente el método “AnularTarea” para cambiar su estado y evitar que sea ejecutada.

```

AddTarea(System.DateTime, string, int, string, string, System.Collections.Generic.List<Tarea>)
AnularTarea(System.DateTime, string)
CopyAll(System.IO.DirectoryInfo, System.IO.DirectoryInfo)
CTarea()
EjecutarTarea(System.Windows.Forms.Timer)
GetTareas()
InsertRutas(System.Data.OleDb.OleDbTransaction, System.DateTime, string, string)
InsertTarea(System.Data.OleDb.OleDbTransaction, System.DateTime, string, string)
CmdCarpetas
CmdTareas
CNN
DACarpetas
DATareas
DS
TablaCarpetas
TablaTareas

```

Observe que han declarado como objetos privados a la clase además de la conexión (CNN) y del DataSet (DS), los objetos Command y los DataAdapter de cada tabla, esto es necesario porque se usarán transacciones y estos objetos deben estar disponibles en varios métodos de la clase.

Veamos ahora la implementación completa de la clase “**CUusuario**”:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data;
using System.Data.OleDb;

namespace SP2
{
    public class CUusuario
    {
        DataSet DS;
        String Tabla = "Usuarios";
        // constructor
        public CUusuario()
        {
            try
            {
                OleDbConnection cnn = new OleDbConnection();
                cnn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0; Data
                Source=BackupProgramado.mdb";
                cnn.Open();
                DS = new DataSet();
                OleDbCommand cmd = new OleDbCommand();
                cmd.Connection = cnn;
                cmd.CommandType = CommandType.TableDirect;
                cmd.CommandText = Tabla;
                OleDbDataAdapter DA = new OleDbDataAdapter(cmd);
                DA.Fill(DS, Tabla);
            }
            catch { }
        }
    }
}

```

```

        DataColumn[] pk = new DataColumn[1];
        pk[0] = DS.Tables[Tabla].Columns["usuario"];
        DS.Tables[Tabla].PrimaryKey = pk;
        OleDbCommandBuilder cb = new OleDbCommandBuilder(DA);
        cnn.Close();
    }
    catch (Exception ex)
    {
        String MsgErr = "CUusuario: " + ex.Message;
        throw new Exception(MsgErr);
    }
}

public DataTable GetUsuarios()
{
    if (DS.Tables.Count == 1)
    {
        return DS.Tables[Tabla];
    }
    else
    {
        throw new Exception("La tabla no existe");
    }
}

public String GetUsuario(int usuario)
{
    String nombre = "";
    DataRow drU = DS.Tables[Tabla].Rows.Find(usuario);
    if(drU != null)
    {
        nombre = drU["Nombre"].ToString();
    }
    return nombre;
}

public String GetPasswordUsuario(int usuario)
{
    DataRow drU = DS.Tables[Tabla].Rows.Find(usuario);
    if(drU != null)
    {
        return drU["palabra"].ToString();
    }
    else
    {
        throw new Exception("El usuario no existe");
    }
}

public void Dispose()
{
    DS.Dispose();
}
}
}

```

El método **constructor** realiza la conexión con la base de datos y carga el dataSet con la tabla de Usuarios, define también la clave primaria de la tabla.

El método “**GetUsuario**” recibe el número de usuario por parámetro y devuelve el nombre del usuario, para buscarlo se usa el método “Find” ya que la tabla tiene definida la clave primaria por el mismo campo del número de usuario.

El método “**GetPasswordUsuario**” funciona igual al anterior, pero devuelve el valor del campo “palabra” (password) del usuario.

El método “**Dispose**” libera los recursos del DataSet.

Implementación de la clase “CTarea”

```
public class CTarea
{
    OleDbConnection CNN;
    DataSet DS;
    OleDbDataAdapter DATareas;
    OleDbDataAdapter DACarpetas;
    OleDbCommand CmdTareas;
    OleDbCommand CmdCarpetas;
    String TablaTareas = "Tareas";
    String TablaCarpetas = "Carpetas";

    public CTarea()
    {
        try
        {
            CNN = new OleDbConnection();
            CNN.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0; Data
Source=BackupProgramado.mdb";
            CNN.Open();
            DS = new DataSet();
            CmdTareas = new OleDbCommand();
            CmdTareas.Connection = CNN;
            CmdTareas.CommandType = CommandType.TableDirect;
            CmdTareas.CommandText = TablaTareas;
            DATareas = new OleDbDataAdapter(CmdTareas);
            DATareas.Fill(DS, TablaTareas);
            DataColumn[] pkT = new DataColumn[2];
            pkT[0] = DS.Tables[TablaTareas].Columns["Fecha"];
            pkT[1] = DS.Tables[TablaTareas].Columns["Hora"];
            DS.Tables[TablaTareas].PrimaryKey = pkT;
            OleDbCommandBuilder cbT = new OleDbCommandBuilder(DATareas);
            //
            CmdCarpetas = new OleDbCommand();
            CmdCarpetas.Connection = CNN;
            CmdCarpetas.CommandType = CommandType.TableDirect;
            CmdCarpetas.CommandText = TablaCarpetas;
            DACarpetas = new OleDbDataAdapter(CmdCarpetas);
            DACarpetas.Fill(DS, TablaCarpetas);
            DataColumn[] pkC = new DataColumn[3];
            pkC[0] = DS.Tables[TablaCarpetas].Columns["Fecha"];
            pkC[1] = DS.Tables[TablaCarpetas].Columns["Hora"];
            pkC[2] = DS.Tables[TablaCarpetas].Columns["Orden"];
```



```

        CmdTareas.Transaction = transaccion;
        DataRow dr = DS.Tables[TablaTareas].NewRow();
        dr["Fecha"] = Fecha;
        dr["Hora"] = Hora;
        dr["Usuario"] = usuario;
        dr["RutaDestino"] = RutaDestino;
        dr["Observacion"] = Observacion;
        dr["Estado"] = 0; // estado pendiente
        DS.Tables[TablaTareas].Rows.Add(dr);
        DATareas.Update(DS, TablaTareas);
    }
    catch(Exception ex)
    {
        throw new Exception(ex.Message);
    }
}

public void InsertRutas(OleDbTransaction transaccion, DateTime Fecha,
                        String Hora, List<String> Rutas)
{
    int orden = 1;
    try
    {
        CmdCarpetas.Transaction = transaccion;
        foreach (String ruta in Rutas)
        {
            DataRow dr = DS.Tables[TablaCarpetas].NewRow();
            dr["Fecha"] = Fecha;
            dr["Hora"] = Hora;
            dr["Orden"] = orden;
            dr["RutaOrigen"] = ruta;
            DS.Tables[TablaCarpetas].Rows.Add(dr);
            orden++; // incrementa el número de orden
        }
        DACarpetas.Update(DS, TablaCarpetas);
    }
    catch(Exception ex)
    {
        throw new Exception(ex.Message);
    }
}

public void AnularTarea(DateTime Fecha,
                        String Hora)
{
    try
    {
        // formar el arreglo con los valores de la clave a buscar en Tareas
        object[] clave = new object[2];
        clave[0] = Fecha; // primer campo de la clave
        clave[1] = Hora; // segundo campo de la clave
        DataRow dr = DS.Tables[TablaTareas].Rows.Find(clave);
        if (dr != null)
        {
            // si encuentra la tarea se controla el estado
            if ((int)dr["Estado"] != 0) // solo se anulan tareas pendientes
            {
                throw new Exception("La tarea no está pendiente, no se puede
anular.");
            }
            dr.BeginEdit();
            dr["Estado"] = 2; // estado anulada
        }
    }
}

```

```

        dr.EndEdit();
        DATareas.Update(DS, TablaTareas);
    }
    else
    {
        // si no encuentra la tarea se dispara una excepción
        throw new Exception("La tarea no existe");
    }
}
catch (Exception ex)
{
    throw ex;
}
}

public void EjecutarTarea(Timer timer)
{
    timer.Enabled = false;
    try
    {
        DateTime Fecha = DateTime.Now.Date;
        String Hora = DateTime.Now.ToShortTimeString().Substring(0, 5);
        Object[] clave = new Object[2];
        clave[0] = Fecha;
        clave[1] = Hora;
        DataRow dr = DS.Tables[TablaTareas].Rows.Find(clave);
        // controlar que exista la tarea y que su estado sea Pendiente: 0
        if (dr != null && (int)dr["Estado"] == 0)
        {
            int orden = 1;
            bool existeRuta = true;
            // buscar todas las carpetas a copiar de la tarea
            while (existeRuta)
            {
                Object[] claveRuta = new Object[3];
                claveRuta[0] = Fecha;
                claveRuta[1] = Hora;
                claveRuta[2] = orden;
                DataRow drR = DS.Tables[TablaCarpetas].Rows.Find(claveRuta);
                if (drR != null)
                {
                    // realizar las copias de los archivos
                    DirectoryInfo diSource = new
DirectoryInfo(drR["RutaOrigen"].ToString());
                    DirectoryInfo diTarget = new
DirectoryInfo(dr["RutaDestino"].ToString());

                    CopyAll(diSource, diTarget); // copia el contenido de la
carpeta

                    orden++;
                }
                else
                {
                    existeRuta = false; // no hay más rutas para copiar
                }
            }
            // actualizar el estado de la tarea
            dr.BeginEdit();
            dr["Estado"] = 1; // estado finalizada
            dr.EndEdit();
            DATareas.Update(DS, TablaTareas);
        }
    }
}

```

```

    }
    catch (Exception ex)
    {
        throw ex;
    }
    timer.Enabled = true;
}

public void CopyAll(DirectoryInfo source, DirectoryInfo target)
{
    String Ruta = Path.Combine(target.FullName, source.Name);
    Directory.CreateDirectory(Ruta); // si existe crea el directorio destino

    // copiar los archivos del directorio origen al directorio destino.
    foreach (FileInfo fi in source.GetFiles())
    {
        fi.CopyTo(Path.Combine(Ruta, fi.Name), true);
    }
}

public void Dispose()
{
    DS.Dispose();
}

}
}

```

El método **constructor** realiza la conexión con la base de datos y carga el DataSet con las tablas de Tareas y Carpetas, define también las claves primarias de cada tabla.

El método “**GetTareas**” devuelve la tabla completa de Tareas.

El método “**AddTarea**” recibe por parámetro todos los valores necesarios para grabar una nueva tarea: fecha, hora, usuario, carpeta destino para las copias, observación y una lista de Strings con las rutas de las carpetas origen para la copia de los archivos. Este método hace uso de una transacción y las operaciones involucradas son las que se ejecutan con los métodos “**InsertTarea**” e “**InsertRutas**”, si ambas se ejecutan correctamente se realiza el Commit y en caso de algún error se realiza el Rollback.

Vemos en detalle la implementación de estos dos últimos métodos.

```

public void InsertTarea(OleDbTransaction transaccion, DateTime Fecha,
                        String Hora,
                        int usuario,
                        String RutaDestino,
                        String Observacion)
{
    try
    {
        CmdTareas.Transaction = transaccion;
    }
}

```

```

        DataRow dr = DS.Tables[TablaTareas].NewRow();
        dr["Fecha"] = Fecha;
        dr["Hora"] = Hora;
        dr["Usuario"] = usuario;
        dr["RutaDestino"] = RutaDestino;
        dr["Observacion"] = Observacion;
        dr["Estado"] = 0; // estado pendiente
        DS.Tables[TablaTareas].Rows.Add(dr);
        DATareas.Update(DS, TablaTareas);
    }
    catch(Exception ex)
    {
        throw new Exception(ex.Message);
    }
}

public void InsertRutas(OleDbTransaction transaccion, DateTime Fecha,
                        String Hora, List<String> Rutas)
{
    int orden = 1;
    try
    {
        CmdCarpetas.Transaction = transaccion;
        foreach (String ruta in Rutas)
        {
            DataRow dr = DS.Tables[TablaCarpetas].NewRow();
            dr["Fecha"] = Fecha;
            dr["Hora"] = Hora;
            dr["Orden"] = orden;
            dr["RutaOrigen"] = ruta;
            DS.Tables[TablaCarpetas].Rows.Add(dr);
            orden++; // incrementa el número de orden
        }
        DACarpetas.Update(DS, TablaCarpetas);
    }
    catch(Exception ex)
    {
        throw new Exception(ex.Message);
    }
}
}

```

Ambos métodos reciben varios parámetros y el primero de ellos es un objeto de tipo **OleDbTransaction**, ese objeto debe ser asignado a la propiedad “**Transaction**” de cada uno de los comandos (OleDbCommand) creados en el constructor de la clase para el manejo de cada tabla. Sin ese valor asignado los cambios realizados en las tablas no podrán ser controlados por la transacción.

El resto de los parámetros que recibe cada método corresponden a los valores que se usarán para crear el nuevo registro en el caso de la tabla Tareas y para crear uno o varios registros en el caso de la tabla de Carpetas, dependiendo de la cantidad de elementos que contenga la lista “Rutas”.

El método “**AnularTarea**” recibe por parámetro la fecha y la hora de la tarea que se desea anular, se realiza una búsqueda con el método “Find” y si el registro es localizado y además su

estado es pendiente se edita y el campo “Estado” recibe el valor 2 indicando que la tarea queda anulada y no se ejecutará.

El método “**EjecutarTarea**” recibe por parámetro el control Timer, necesario para poder detenerlo y reanudarlo al final del proceso. Se debe detener para que no se vuelva a ejecutar hasta que no termine completamente de realizar las copias de los archivos.

Luego toma la fecha y hora actuales del sistema y con esos valores busca en la tabla de Tareas alguna tarea que esté grabada con esos mismos valores de fecha y hora y que además esté en estado pendiente de ejecutar, si encuentra una tarea para ejecutar comienza un ciclo donde busca en la tabla de Carpetas la ruta origen de las copias a realizar, con todos esos valores conocidos se ejecuta la copia de archivos por medio del método “CopyAll”, el proceso se repite con el resto de carpetas que tenga la tarea. Finalmente, el registro de la tarea se edita y se modifica el estado pasando de pendiente (0) a terminada (1).

En el método “**CopyAll**” se hace uso de las clases “DirectoryInfo”, Path”, “Directory” y “FileInfo”, todas definidas en el espacio “System.IO” y ya conocidas en la materia Laboratorio de Programación 1 y 2.

Referencias a estas 4 clases:

<https://learn.microsoft.com/en-us/dotnet/api/system.io.directoryinfo?view=netframework-4.8.1>

<https://learn.microsoft.com/en-us/dotnet/api/system.io.path?view=netframework-4.8.1>

<https://learn.microsoft.com/en-us/dotnet/api/system.io.directory?view=netframework-4.8.1>

<https://learn.microsoft.com/en-us/dotnet/api/system.io.fileinfo?view=netframework-4.8.1>

Finalmente, el método “**Dispose**” libera los recursos usado por el objeto DataSet.

Continuamos ahora con la implementación de los eventos del primer formulario (Form1), cuyo código completo resulta así:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```

using System.Windows.Forms;
using System.Diagnostics;

namespace SP2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            private void Form1_Load(object sender, EventArgs e)
            {
                // cargar los usuarios
                CUsuario usr = new CUsuario();
                cmbUsuarios.DisplayMember = "Nombre";
                cmbUsuarios.ValueMember = "Usuario";
                cmbUsuarios.DataSource = usr.GetUsuarios();
                // iniciar el timer
                tmrTareas.Interval = 30000;
                tmrTareas.Enabled = true;
            }

            private void tmrTareas_Tick(object sender, EventArgs e)
            {
                try
                {
                    CTarea tarea = new CTarea();
                    tarea.EjecutarTarea(tmrTareas);
                }
                catch (Exception ex)
                {
                    MessageBox.Show(ex.Message);
                }
            }

            private bool ValidarPassword()
            {
                bool resultado = false;
                CUsuario usr = new CUsuario();
                String password = usr.GetPasswordUsuario((int)cmbUsuarios.SelectedValue);
                if (password.CompareTo(txtPassword.Text) == 0)
                {
                    resultado = true;
                }
                return resultado;
            }

            private bool ValidarDatos()
            {
                DateTime fecha = dtpFecha.Value.Date;
                String hora = dtpHora.Value.ToShortTimeString().Substring(0,5);
                if (fecha.CompareTo(DateTime.Now.Date) < 0)
                {
                    throw new Exception("La fecha no puede ser menor a la fecha actual");
                }
                if (fecha.CompareTo(DateTime.Now.Date) == 0 &&
                    hora.CompareTo(DateTime.Now.ToString("HH:mm")) < 0)
                {
                    throw new Exception("La hora no puede ser manor a la hora actual");
                }
            }
        }
    }
}

```

```

        if(txtRutaDestino.Text == "")
        {
            throw new Exception("Debe seleccionar la ruta destino de las copias");
        }
        if (lstCarpetas.Items.Count == 0)
        {
            throw new Exception("Debe seleccionar al menos una carpeta para
copiar");
        }
        return true;
    }

    private void InicializarInterfaz()
    {
        txtPassword.Text = "";
        txtRutaDestino.Text = "";
        lstCarpetas.Items.Clear();
        txtObservacion.Text = "";
    }

    private void btnAceptarTarea_Click(object sender, EventArgs e)
    {
        try
        {
            if (ValidarPassword())
            {
                if (ValidarDatos())
                {
                    CTarea tarea = new CTarea();
                    // crear una lista con los nombres de las carpetas a copiar
                    List<String> lista = new List<string>();
                    for (int i = 0; i < lstCarpetas.Items.Count; i++)
                    {
                        lista.Add(lstCarpetas.Items[i].ToString());
                    }
                    DateTime fecha = dtpFecha.Value.Date;
                    // la hora se guarda en formato hora:minutos,
                    // con la hora en formato 24 horas
                    String hora = dtpHora.Value.ToString("HH:mm");
                    // obtener el número del usuario seleccionado en el comboBox
                    int usuario = (int)cmbUsuarios.SelectedValue;
                    String rutaDestino = txtRutaDestino.Text;
                    String observacion = txtObservacion.Text;
                    tarea.AddTarea(fecha, hora, usuario, rutaDestino, observacion,
lista);

                    // restablecer la interfaz
                    InicializarInterfaz();
                }
            }
            else
            {
                MessageBox.Show("Password incorrecto!.");
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }

    private void btnElegir_Click(object sender, EventArgs e)
    {

```

```

        txtRutaDestino.Text = "";
        FolderBrowserDialog dlg = new FolderBrowserDialog();
        dlg.Description = "Seleccione la carpeta destino";
        if (dlg.ShowDialog() == DialogResult.OK)
        {
            txtRutaDestino.Text = dlg.SelectedPath;
        }
        dlg.Dispose();
    }

    private void cmbUsuarios_SelectedIndexChanged(object sender, EventArgs e)
    {
        txtPassword.Text = "";
    }

    private void btnAgregar_Click(object sender, EventArgs e)
    {
        FolderBrowserDialog dlg = new FolderBrowserDialog();
        dlg.Description = "Seleccione la carpeta a copiar";
        if (dlg.ShowDialog() == DialogResult.OK)
        {
            lstCarpetas.Items.Add(dlg.SelectedPath);
        }
        dlg.Dispose();
    }

    private void btnQuitar_Click(object sender, EventArgs e)
    {
        if (lstCarpetas.SelectedIndex != -1)
        {
            lstCarpetas.Items.RemoveAt(lstCarpetas.SelectedIndex);
        }
    }

    private void btnVerTareas_Click(object sender, EventArgs e)
    {
        Form2 dlg = new Form2();
        dlg.ShowDialog();
    }
}

```

En el evento “**Load**” se inicia la ejecución del Timer, asignado un intervalo de 30 segundos.

En el evento “**Tick**” del timer se crea un objeto de tipo CTarea y se invoca al método “EjecutarTarea” para controlar y ejecutar la tarea de copiar los archivos de backup si hubiera alguna programada para la fecha y hora actual.

Los métodos “**ValidarPassword**” y “**ValidarDatos**” controlan que toda la información ingresada en el formulario sea correcta y válida para poder grabarla en la base de datos, las principales validaciones son el password del usuario seleccionado y que la fecha y hora para la

nueva tarea correspondan a un tiempo futuro. Los errores detectados generan excepciones y no permitirán grabar ningún registro.

El método “**InicializarInterfaz**” se ocupa de colocar los controles del formulario en su estado inicial.

El evento “Click” del botón “**AceptarTarea**” ejecuta las validaciones de datos que comentamos anteriormente y si todo es correcto hace uso de un objeto de la clase “CTarea” invocando el método “AddTarea” pasando por parámetro todos los valores ingresados por el usuario en los controles del formulario. Finalmente ejecuta el método “InicializarInterfaz” para restaurar el estado del formulario.

En el evento “Click” del botón “**Elegir**” se utiliza un “FolderBrowserDialog” para que el usuario pueda seleccionar la carpeta destino de las copias backup, el valor obtenido se asigna al control TextBox llamado “txtRutaDestino”.

En el evento “Click” del botón “**Agregar**” también se trabaja con un “FolderBrowserDialog” pero la ruta obtenida de cada carpeta a incluir en el backup se va agregando al control ListBox de nombre “lstCarpetas”.

En el caso del evento “Click” del botón “**Quitar**”, se controla que exista algún ítem del ListBox seleccionado para luego eliminarlo.

Finalmente, en el evento “Click” del botón “**VerTareas**” se crea un objeto de la clase “Form2” y se ejecuta el método “ShowDialog” para que el usuario pueda trabajar con él.

Y para finalizar con el código veamos la implementación del segundo formulario: **Form2**

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

namespace SP2
{
    public partial class Form2 : Form
    {

```

```

public Form2()
{
    InitializeComponent();
}

private void Form2_Load(object sender, EventArgs e)
{
    CargarTareas();
}

private void CargarTareas()
{
    grTareas.Rows.Clear();
    // cargar la grilla con los datos de las tareas
    String[] Estados = new String[3] { "Pendiente", "Terminada", "Anulada" };
    CUsuario usr = new CUsuario();
    CTarea tarea = new CTarea();
    DataTable TablaTarea = tarea.GetTareas();
    foreach (DataRow dr in TablaTarea.Rows)
    {
        String Nombre = usr.GetUsuario((int)dr["usuario"]);
        grTareas.Rows.Add(dr["Fecha"].ToString().Substring(0, 10),
dr["Hora"].ToString(),
        Nombre, Estados[(int)dr["Estado"]]);
    }
}

private void btnAnular_Click(object sender, EventArgs e)
{
    if(grTareas.SelectedRows.Count == 1)
    {
        try
        {
            DateTime fecha =
DateTime.Parse(grTareas.SelectedRows[0].Cells[0].Value.ToString());
            String hora = grTareas.SelectedRows[0].Cells[1].Value.ToString();
            CTarea tarea = new CTarea();
            tarea.AnularTarea(fecha, hora);
            CargarTareas();
        }
        catch(Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
    else
    {
        MessageBox.Show("Debe seleccionar una tarea para anular");
    }
}

private void btnVolver_Click(object sender, EventArgs e)
{
    Close();
}
}
}

```

Desde el evento “**Load**” se ejecuta el método “**CargarTareas**” que se encarga de obtener los datos de la tabla Tareas y darles el formato correcto para mostrarlos en la grilla (control DataGridView), Utiliza las clases “CTareas” y “CUusuarios” y un arreglo de tipo string para mostrar los nombres de los estados en lugar de sus valores numéricos.

El evento “Click” del botón “**AnularTarea**” invoca el método “**AnularTarea**” del objeto tarea pasando por parámetro los valores de fecha y hora de la tarea que el usuario seleccionó en la grilla. Si no hay ninguna tarea seleccionada el botón no realiza ninguna acción.

Finalmente, el botón “**Volver**” cierra el formulario y retorna la ejecución al formulario principal (Form1).

De esta forma finalizamos la resolución de la situación profesional solicitada, sugerimos ejecutarla y verificar que las tareas de copias son ejecutadas correctamente en los horarios programados y que la carpeta destino contenga todos los archivos.

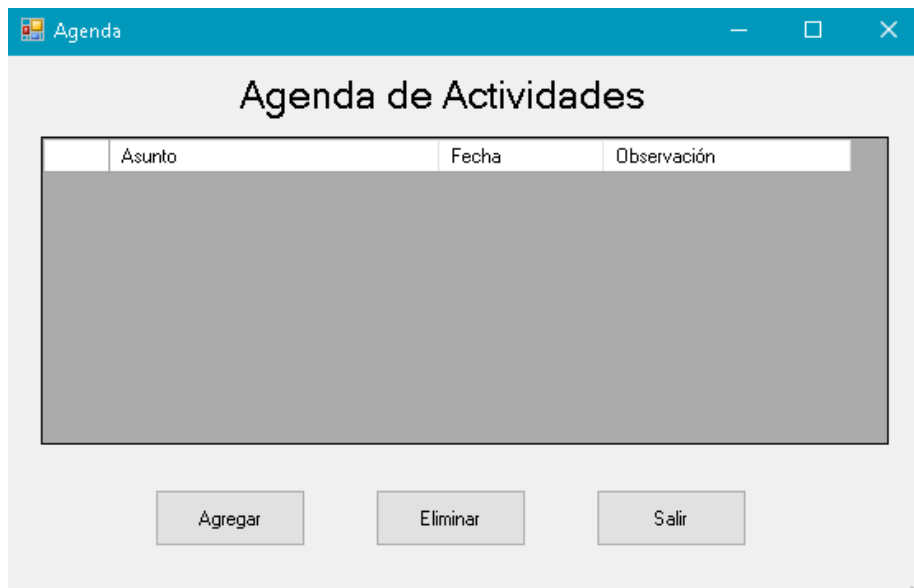
SP2/Ejercicio por resolver

La consultora para la que trabaja le solicita a usted que desarrolle una aplicación que permita a sus usuarios crear una agenda de actividades y establecer avisos de vencimientos. Los datos se registran en una base de datos tipo Access que cuenta con una tabla denominada Actividades, cuya estructura se detalla a continuación:

- IdActividad	Integer
- Asunto	Text (50)
- Fecha	Fecha/Hora
- Observación	Text (30)

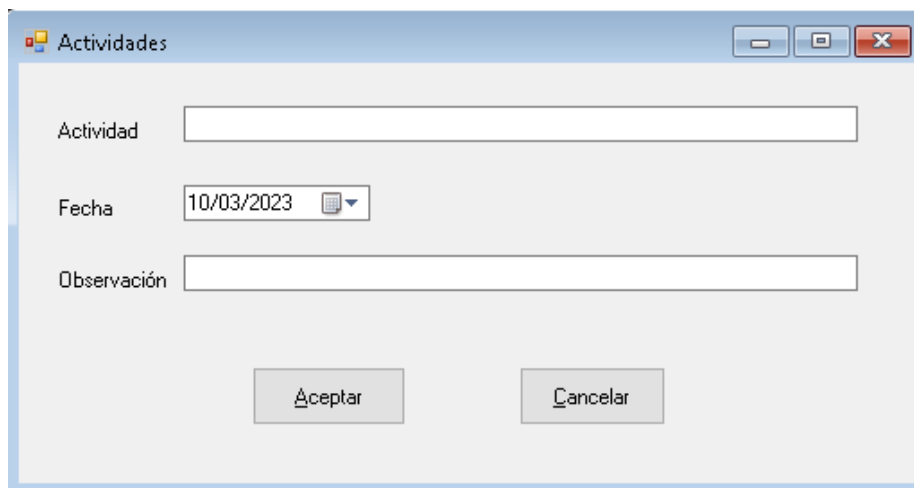
El campo IdActividad es la clave primaria de la tabla.

Al iniciar la aplicación debe presentar la siguiente interfaz gráfica:



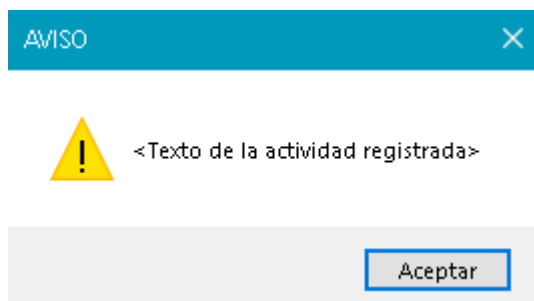
El botón “Agregar” abrirá un segundo formulario para el ingreso de nuevas actividades, el botón “Eliminar” debe permitir eliminar la actividad seleccionada en la grilla y el botón “Salir” cierra la aplicación.

Para agregar las tareas:



Se validará que el campo Actividad no esté vacío y que la fecha sea un valor futuro, el valor del campo Observación es opcional. La fecha a registrar será la seleccionada en el control DateTimePicker con la hora siempre en el valor “12:00:00”. Ese será el vencimiento de la tarea.

Y para mostrar un aviso programado:



El aviso debe mostrarse automáticamente 2 veces para cada actividad, la primera vez se mostrará 24 horas antes y la segunda vez se mostrará 12 horas antes del vencimiento registrado de la tarea.

Este ejercicio deberá ser resuelto en forma práctica creando un proyecto en Visual Studio.

1. Indique la opción correcta.

El control DateTimePicker se usa tanto para seleccionar fechas como para seleccionar horas.

Verdadero ☒ X

Falso

2. Indique la opción correcta.

El control DateTimePicker no permite seleccionar una fecha y una hora al mismo tiempo

Verdadero

Falso ☒ X

3. Indique la opción correcta.

En una transacción se puede trabajar como máximo con dos tablas de la base de datos

Verdadero

Falso ☒ X

4. Indique la opción correcta.

Una transacción en ADO .NET sólo puede controlar operaciones sobre la base de datos.

Vardadero ☒ X

Falso

5. Indique la opción correcta.

En un control Timer el evento a programar para ejecutar las acciones periódicas del timer es:

Timer

Tick ☒ X

Load

Exceute

6. Indique la opción correcta.

En un control Timer el valor del intervalo de ejecución está establecido en segundos

Verdadero

Falso X