Final LUG

Lenguajes De Última Generación (Universidad Abierta Interamericana)

Ensamblados:

Aplicaciones NET: Al compilar una aplicación .NET se obtiene un módulo ensamblado. Puede ser un EXE o una DLL dependiendo el tipo de proyecto que se desarrolle.

Un ensamblado puede estar formado por uno o varios módulos. Una característica importante de estos módulos es que pueden ser *autosuficientes o dependientes* de otros módulos.

La composición de un ensamblado se agrupa en:

- Encabezados de archivos Windows PE
- Encabezado de archivo .Net framework
- Metadatos
- Lenguaje intermedio MSIL

1. Encabezado: Todos los módulos y archivos EXE tienen un encabezado PE. Una referencia importante que tiene es el código que se tiene que ejecutar cuando el SO los invoca.

Los módulos tienen código MSIL, que son un conjunto de instrucciones nativas comprendidas por la CPU destino. Es por esto, que la primera instrucción de los módulos administrados es JMP (jump) que apunta al código MSIL.

2. Metadatos: descripción de los datos. Describe todas las clases, métodos e interfaces públicos que un ejecutable expone. Describen los tipos a los cuales hace referencia el modulo actual. *No existe forma de entregar un modulo sin metadatos o metadatos sin el módulo.*

3. Lenguaje Intermedio: MSIL □ lenguaje intermedio de Microsoft, una ventaja es que TODOS los lenguajes usados por Microsoft compilan en MSIL. MSIL tiene 2 beneficios:

1. Proporciona un lenguaje intermedio de fácil adaptación a las CPU del mercado. Estas deben tener instalado el framework .NET en la PC donde se instale el programa.
2. Refuerza en cuanto a la seguridad en la ejecución de la aplicación al hacer más robusto al código administrado.

MSIL no solo unifica los lenguajes de alto nivel, sino que también es reversible y puede por ejemplo convertir un programa de vb.net en C# o viceversa.

Ensamblados: Conjunto de uno o mas módulos. Es la mejor unidad de reusó en .NET, por ello se debe mantener junto a los módulos que se complementan entre sí.

- Control de versiones: aspecto importante para un correcto control de las aplicaciones.
- Ensamblados estáticos y dinámicos. La mayoría de los compiladores de C# usan estáticos, en donde estos se corresponden con uno o varios archivos físicos. En cambio, los dinámicos, se crean en la RAM al momento de ser utilizados.
- Manifiesto: archivo que está dentro del ensamblado. Contiene los metadatos del ensamblado. Datos como: versión, referencia cultural y la clave publica de la compañía que lo desarrolla y la lista de sistemas operativos y CPU donde se podrá ejecutar el ensamblado. También tiene una colección de tablas que muestra la lista de archivos que forman parte del ensamblado.

Por defecto el runtime .NET busca los ensamblados que se encuentran en el mismo espacio de directorios de la aplicación que efectúa la llamada, para evitar conflictos con nombres de los archivos. Puede suceder que, sobre todo cuando los ensamblados se comparten con otras apps, que los nombres entre en conflicto, y para asegurar esto se puede generar una clave

publica y privada aleatoria con la aplicación SN y luego firmar el ensamblado de la app, asegurando un nombre único.

Herramientas: el framework .NET ofrece varias herramientas que facilitan la lectura y administración de los ensamblados.

FUSLOGVW: Facilita la lectura de la localización de los ensamblados. Es una herramienta muy importante cuando es necesario seguir errores ocurridos en los ensamblados.

Compilador de línea de comando: se puede desarrollar aplicaciones .NET desde un block de notas y luego compilarse con el VBC.exe. Ej. Se puede crear un EXE desde un archivo.

GACUTIL: permite administrar el cache de ensamblados global (GAC) desde la línea de comandos. Al registrarlo en la GAC una aplicación podrá usar un ensamblado que no se encuentra en su carpeta.

ILDASM: herramienta que permite ver el código administrado.

APPDOMAIN: todas las aplicaciones de NET corren en un APPDOMAIN. La clase APPDOMAIN nos permite conocer detalles e información acerca del dominio donde se encuentra la aplicación. Esta clase tiene un conjunto de propiedades, métodos y eventos muy útiles.

Ej. Se puede conocer el directorio donde se encuentra la aplicación, datos muy importantes a la hora de manejar archivos locales.

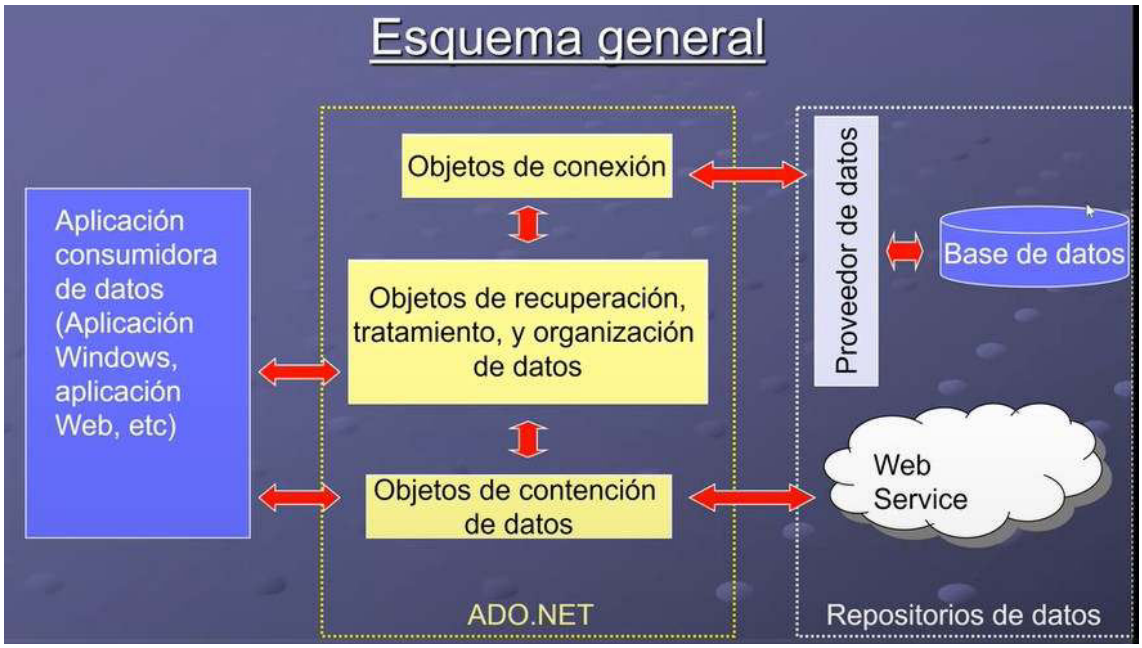
Creación de un APPDOMAIN: al APPDOMAIN se le debe aplicar un ensamblado para poder ejecutarse, al crear un APPDOMAIN va a estar acompañado de un ExecuteAssembly.

Conclusión: no importa que lenguaje se use en el desarrollo, al final todo termina siendo código MSIL o IL. Esto en principio es una gran ventaja, pero genera mucha vulnerabilidad puesto que el código MSIL no es compilado y se puede leer fácilmente con una herramienta.

Esta debilidad también se debe a que es de código abierto, accesible por todos. Se puede liberar o no el código fuente, pero desde el MSIL se puede hacer una reingeniería y obtenerlo igual.

LUG – Clase 1:

ADO.NET es un conjunto de clases del framework que permiten recuperar y manipular repositorios de datos. (distintas bases de datos, SQL Server, PostgreSQL u otras fuentes, archivos, XML, etc.).



ADON.NET es un intermediario entre el programa consumidor de datos y el repositorio de estos. Para trabajar en el medio y comunicarse con una base de datos lo hace mediante un Data Provider o proveedor de datos.

Proveedores de datos:

- Proporciona una interfaz para interactuar con una base de datos.
- Existen distintos tipos de proveedores de datos específicos según el uso.
- Provee una colección de objetos de conexión y recuperación y tratamiento de datos específicos para cada proveedor de datos.

ADO.NET tiene disponible los siguientes proveedores: SQL Server, OLE DB, ODCB, Oracle y se pueden incorporar otros.

1. Objetos de conexión: Este objeto proporciona métodos para conectarse a una base de datos y es especifico según el proveedor de datos. Ej. Para **SQL** Server: **SqlConnection**, para **Oralce**: **OracleConnection**, etc.

2. Objetos de recuperación y tratamiento: permiten interactuar con la base de datos y ejecutar comandos para seleccionar datos, organizarlos, agregar nuevos, modificarlos o borrarlos. Objetos:

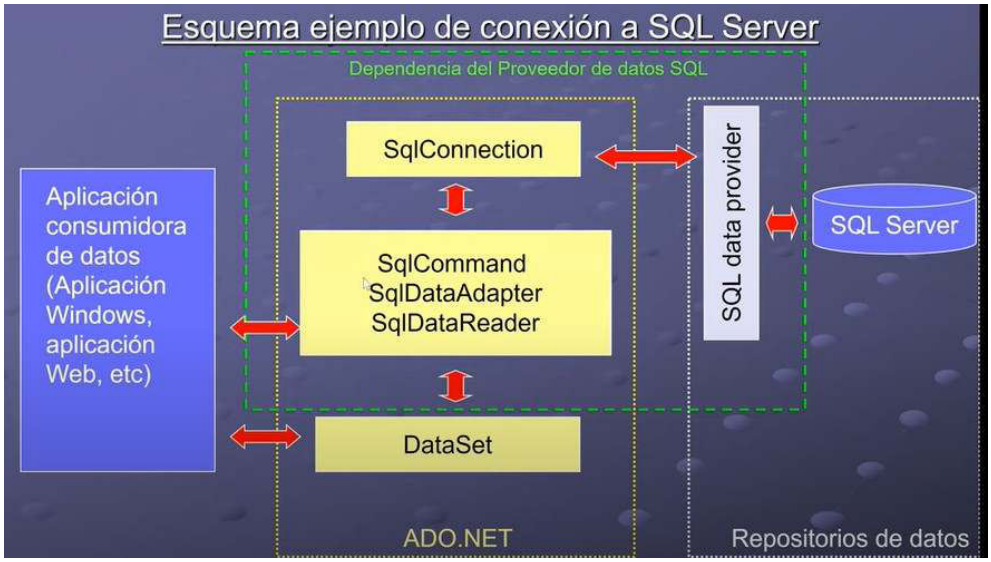
- **Command (SqlCommand)**
- **DataAdapter (SqlDataAdapter)**
- **DataReader (SqlDataReader)**

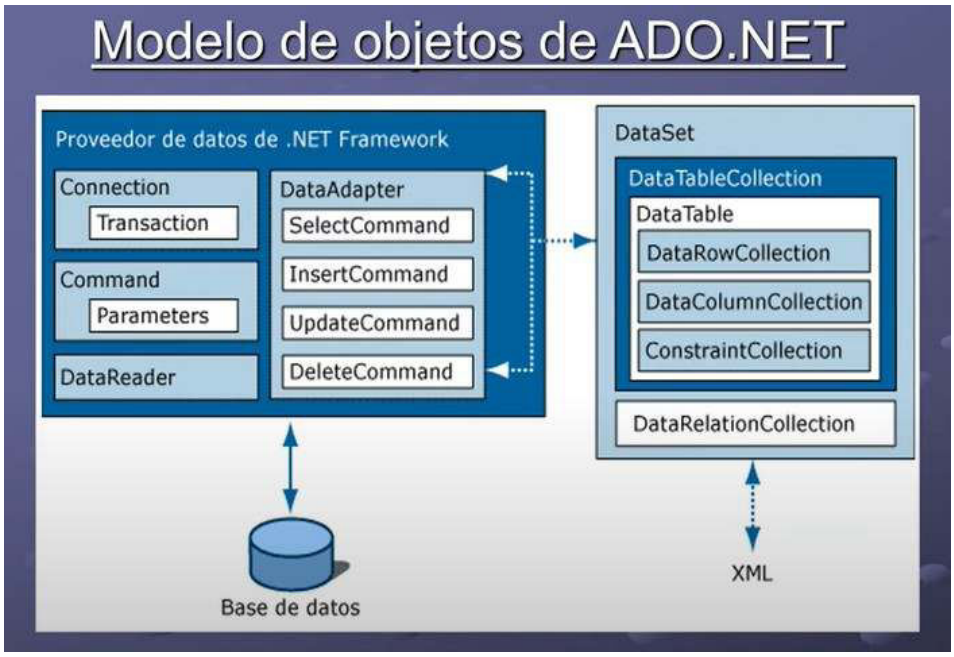
Estos son específicos para SQL. Cambia según el Data Provider.

3. Objetos de contención de datos: permite almacenar en memoria conjunto de datos para pasarlos desde y hacia la aplicación ya sea para leerlos o modificarlos: DataSet, DataTable, DataRow, DataColumn, DataRelation, Constraint.

Básicamente es DataSet y tiene varias cosas dentro, en jerarquía.

*ESTOS OBJETOS son **independientes** del Provider, son generales.*





Espacio de nombres: las clases de ADO.NET se encuentran en el .NET Framework en los siguientes espacios de nombres:

- System.Data (objetos no dependientes de un proveedor de datos).
- System.Data.Common (clases virtuales o interfaces que comparten los proveedores)
- System.Data.SqlClinet (objetos de ejemplo para el proveedor de datos de SQL Server).

Se puede hacer un Data Provider propio o de terceros y ponerlo a disposición para que lo descarguen. Para esto hago mis propias clases y hago que hereden de las clases Data.Common,

así mis clases pueden interactuar con .Net como ej. Usando los métodos Open() y Close(), que son obligatorios y también polimórficos.

Independencia de la base de datos: ADO.NET puede trabajar de 2 formas:

1. Modo conectado: ADO.NET gestiona una conexión y no la cierra hasta terminar de efectuar los comandos contra la base.
2. Modo desconectado: ADO.NET gestiona una conexión para cerrarla inmediatamente después de obtener los datos necesarios. Luego se volverá a conectar si realizo algún cambio en los datos y desea persistirlos en la base.

Ventajas y desventajas de cada uno:

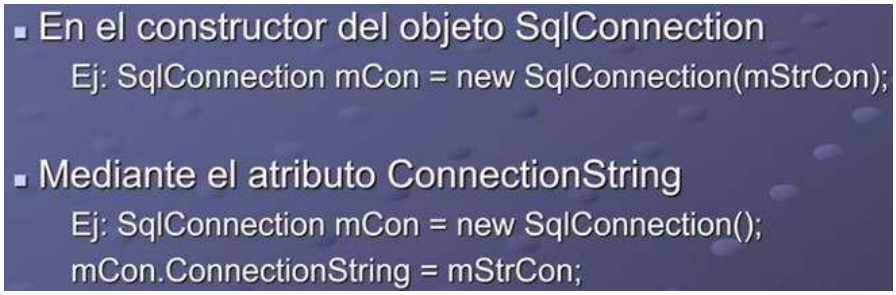
- **Modo conectado**: más eficiente si se busca leer algo rápido y llenar una grilla, una lectura de forma secuencial. (*abre conexión, trabaja con el repo de datos y se desconecta*).
- **Modo desconectado**: para reordenar, buscar, filtrar, modificar o mantener datos en memoria este modo es mejor, ya que ahorra tiempo y recursos de conexión a la BD. (*se conecta lo mas tarde posible y se desconecta lo mas temprano posible, trabaja en memoria, desconectado con un DataSet*). Una ventaja es que no mantengo la conexión prendida todo el tiempo, pero una desventaja es que hay que tener un control de concurrencia. Ya que se traen datos a memoria, se manipulan y luego se deben subir.

1. **Objeto Connection**: permite gestionar una conexión con una base de datos. Para SQL es Sql Connection:

El atributo fundamental de un objeto connection es "**ConnectionString**": Este representa una cadena de texto con la información necesaria que permitirá al objeto Connection conectarse a la BD. Los elementos del string de conexión son:

- Provider: proveedor de datos.
- DataSource: ubicación del servidor de la BD.
- Usuario y clave: credenciales para loguearse.
- InitialCatalog: se coloca el nombre de la BD.

El string de conexión se puede indicar de dos maneras al objeto de conexión:

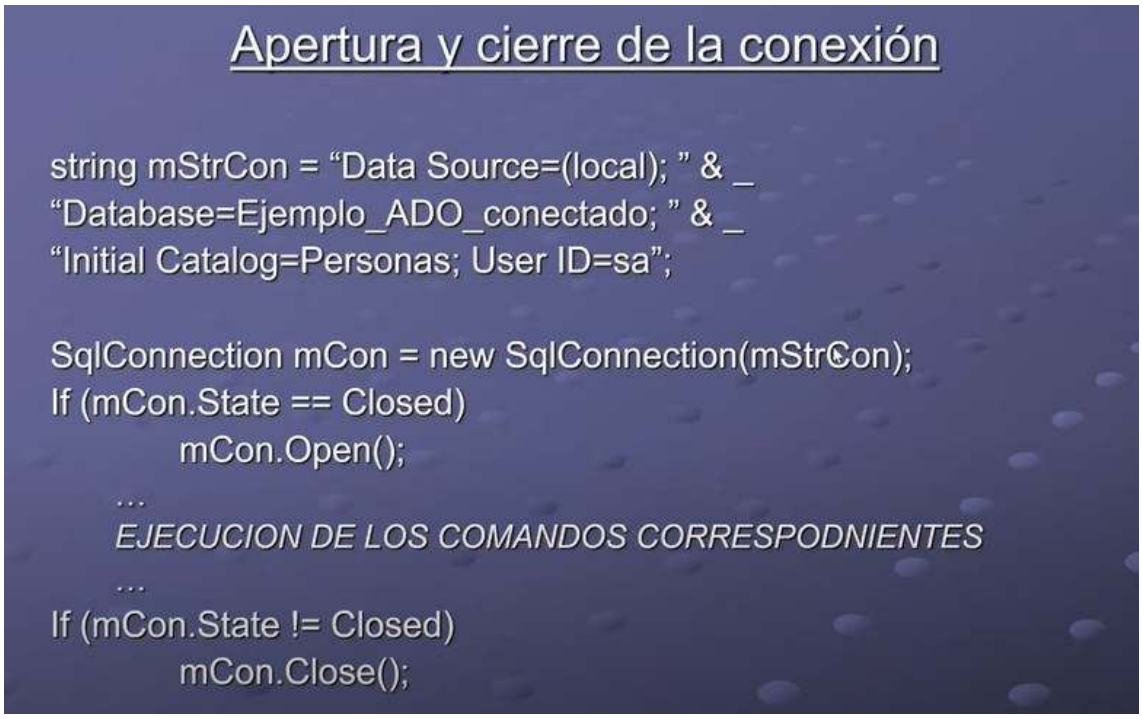


Principales atributos:

- State: indica el estado de la conexión (closed, open, connecting, executing, fetching, broken).
- Database, DataSource, ServerVersion, ConnectionTimeOut. Son que devuelven información de la conexión y servidor.

Principales métodos:

- Open (abre la conexión).
- Close (cierra la conexión).
- BeginTransaction: indica una transacción para la conexión.
- ChangeDatabase: cambia la base de datos activa, sin cerrar la otra.
- CreateCommand: crea un objeto command asociado a esa conexión.



2. OBJETO COMMAND: permite ejecutar comandos contra la base de datos. Estos comandos pueden ser de consultas (SELECT) o manipulación de datos (INSERT, DELETE, UPDATE).

Principales atributos:

- CommandText: es el texto del comando.
- CommandType: indica el tipo del CommandText, si es una sentencia SQL o una Stored Porcedure (función programada y almacenada en una BD).
- Connection: indica la conexión que usara el command para ejecutarse.
- Transaction: indica la transacción asociada al comando.
- Parameters: parámetros asociados al comando.

Principales métodos:

- ExecuteNonQuery: ejecuta una acción que no devuelve resultados, solamente el número de filas afectadas.
- ExecuteReader: ejecuta una consulta de selección y devuelve el DataReader necesario para recorrer los resultados.

- ExecuteScalar: ejecuta una consulta y devuelve un solo valor (primera columna de la primera fila)-
- ExecuteXMLReadres: permite leer el resultado de una consulta “FOR XML”.
- Cancel: cancela la ejecución de un comando.
- CreateParameter: crea un parámetro asociado al comando.

3. Objeto DataReader:

1. Permite iterar los resultados de una consulta SQL.
2. Se usa en modo conectado, por lo tanto, al finalizar su utilización se debe cerrar.
3. Mientras una conexión este sirviendo a un DataReader no se podrá utilizar para ningún otro fin. Solo podrá invocarse su método Close().

Es un puntero al set de resultados. Me deja hacer una lectura secuencial de los resultados, sin tener que traerlo a memoria.

Principales atributos:

- IsClosed**:** indica si el DataReader está cerrado o no.
- FieldCount**:** devuelve la cantidad de columnas que tiene el set de resultados.
- Depth**:** devuelve la cantidad de set de resultados obtenidos. (se puede ejecutar más de una consulta SQL, en la misma instrucción).
- Item**:** permite acceder al elemento indicado por su parámetro.

Principales métodos:

- Read: va avanzando a la siguiente fila de los resultados y devuelve True si hay más filas. (es siempre secuencial, no vuelve atras)
- Close: cierra el DataReader.
- NextResult: avanza al siguiente set de resultados y devuelve True si hay más.
- GetValue: devuelve el valor de la columna con el índice indicado.

```
SqlConnection mCon = new SqlConnection(mStrCon);

mCon.Open();

SqlCommand mCom = new SqlCommand("SELECT
Persona_nombre FROM Persona", mCon);

SqlDataReader mDr = mCom.ExecuteReader();

while(mDr.Read())
   mGrilla.Rows.Add(mDr["Persona_nombre"]);

mDr.Close();
mCon.Close();
```

Objeto DataReader no TIENE CONSTRUCTOR público. Se instancia con el *ExecuteReader*.

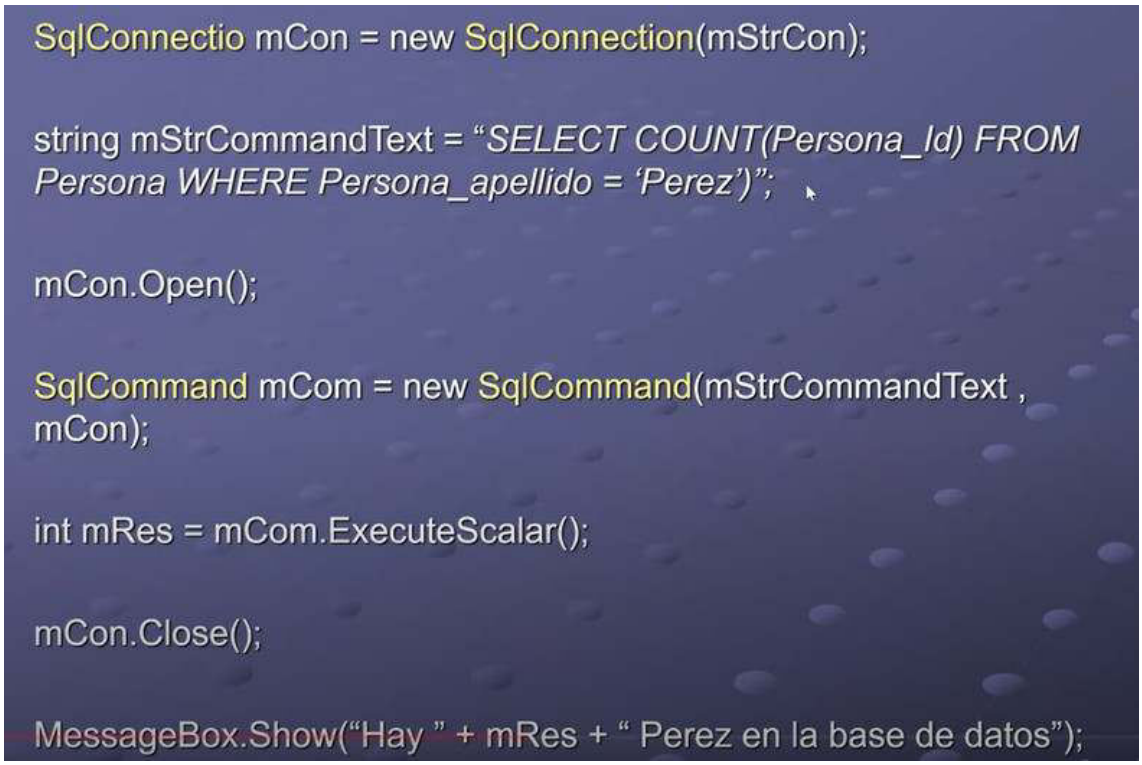
INSERCIÓN:

```
SqlConnection mCon = new SqlConnection(mStrCon);

string mStrCommandText  = "INSERT INTO Persona
(Persona_nombre, Persona_apellido) VALUES ('Pedro', 'Perez')";

mCon.Open();

SqlCommand mCom = new SqlCommand(mStrCommandText ,
mCon);

mCom.ExecuteNonQuery();

mCon.Close();
```

BORRADO:

```
SqlConnection mCon = new SqlConnection(mStrCon);

string mStrCommandText = "DELETE FROM Persona WHERE
Persona_apellido = 'Perez')";

mCon.Open();

SqlCommand mCom = new SqlCommand(mStrCommandText ,
mCon);

int mRes = mCom.ExecuteNonQuery();

mCon.Close();

MessageBox.Show("Se han borrado " + mRes + " registros");
```

LECTURA ESCALAR



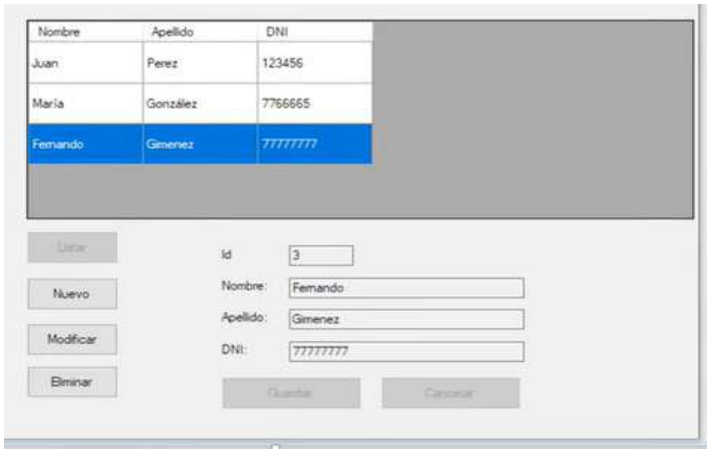
Id autonumérico: se puede hacer desde la base de datos. Ej. tabla Persona, a la columna del id, se puede en propiedades modificar el *Identity Increment y Identity Seed*.

Profe no le gusta esto, prefiere manejarlo desde el código.

```
string mCommandText = "INSERT INTO Persona (Persona_Id, Persona_Nombre, Persona_Apellido, Persona_DNI)
SqlCommand mCom = new SqlCommand(mCommandText, mCon);
mCom.Parameters.AddWithValue("@Persona_Id", ObtenerProximoId());
mCom.Parameters.AddWithValue("@Persona_Nombre", txtNombre.Text);
mCom.Parameters.AddWithValue("@Persona_Apellido", txtApellido.Text);
mCom.Parameters.AddWithValue("@Persona_DNI", txtDNI.Text);

mCon.Open();
mCom.ExecuteNonQuery();
mCon.Close();
```

Así se le pasan parámetros.

Para mejorarlo se puede agregar un try, catch y un bloque finally para cerrar la conexión.

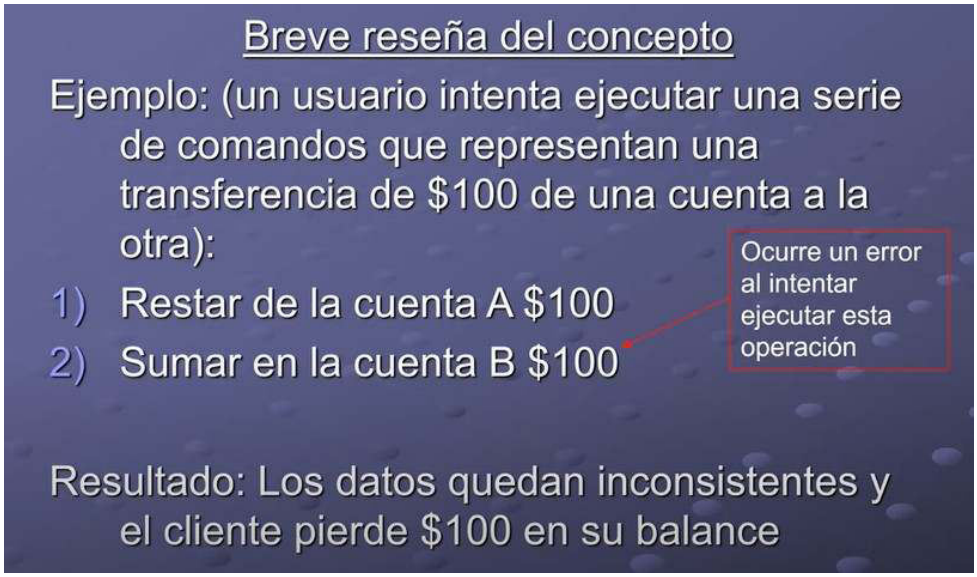


Cuando se selecciona una fila se carga los textbox. Para eso en cada click de la grilla busca ese registro a la BD y no lo trae directo de la grilla. Esto por 2 cosas:

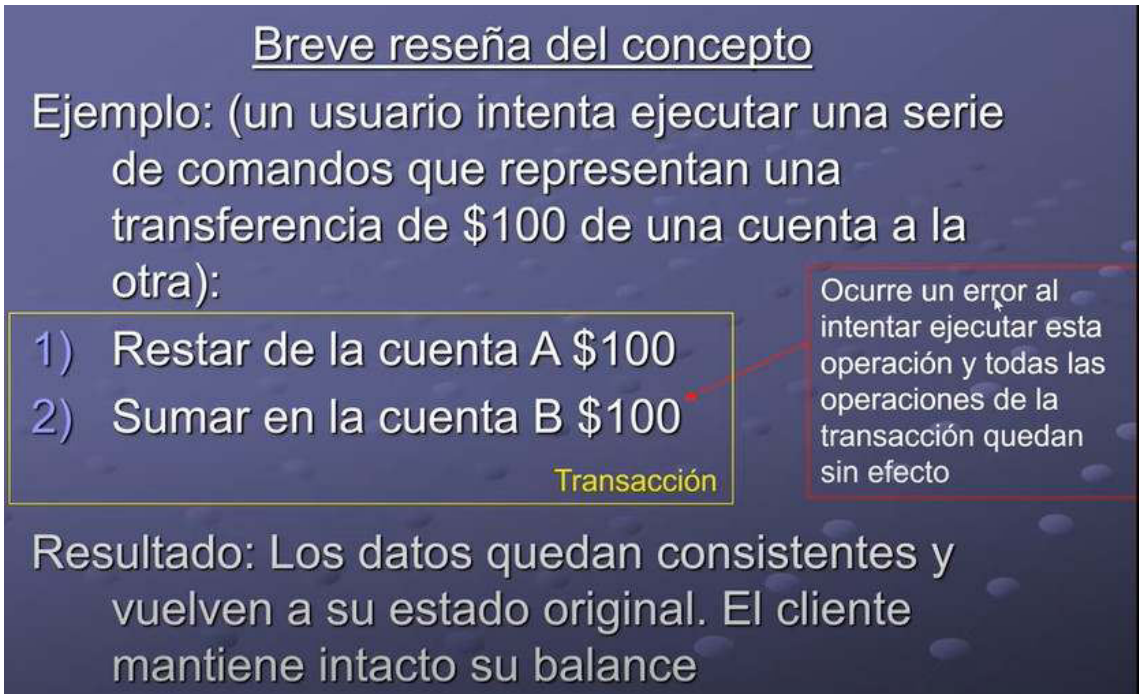
1. La grilla por ahí muestra algunas columnas de las que en realidad tiene la tabla.
2. Existirá una lógica de negocio en BE, y siempre hay que pasar por ahí. Si se trae directo de la grilla se estaría salteando a la BE y a la lógica de negocio y eso es algo malo, hay que pasar por la capa de negocio.

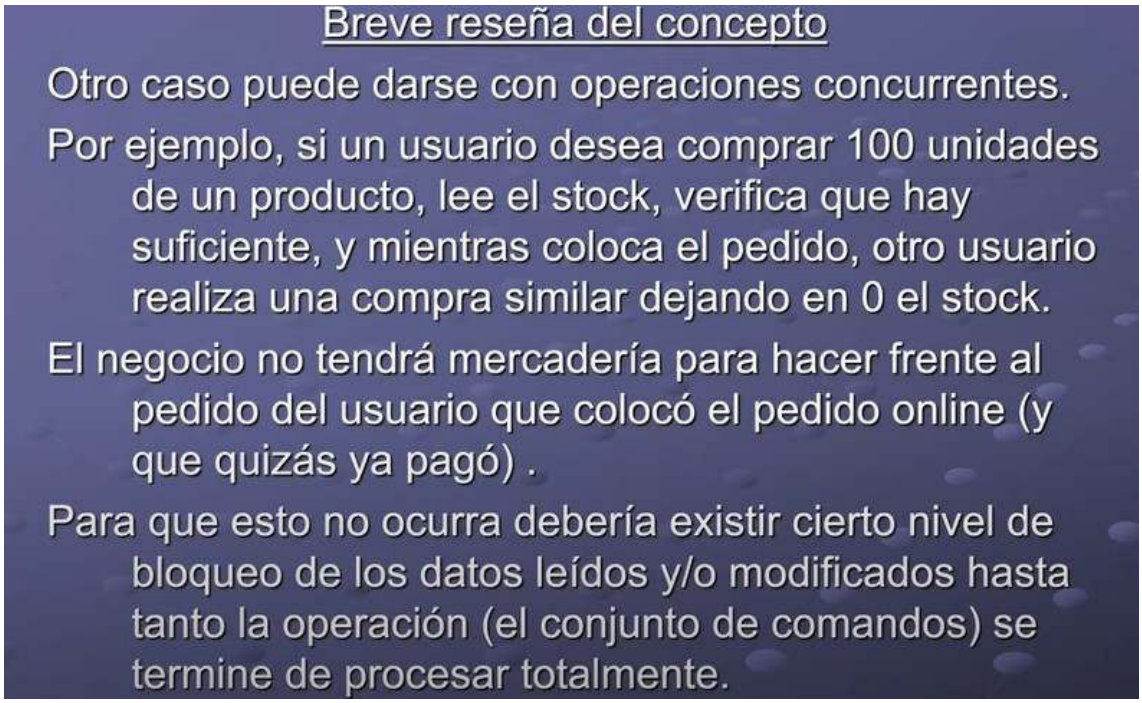
TRANSACCIONES: marco donde voy a encuadrar varias operaciones, más de una. Una transacción garantiza:

1- La atomicidad de una serie de comandos ejecutados contra una base de datos (atomicidad = átomos que no se pueden dividir, si hago 5 operaciones en el marco de una transacción no se pueden dividir, SE EJECUTA TODO).
2- La consistencia de la información.
3- El aislamiento de las operaciones.



Para garantizarlo, se engloba todo en una transacción.





Garantiza un aislamiento, hasta que no termine mi operación no puede llegar otra.

Transacciones: ADO.NET provee soporte para transacciones mediante el objeto Transaction. Este objeto es dependiente del proveedor de datos. Para SQL es **SqlTransaction**.

Una transacción se inicia desde el objeto de conexión con el método **BeginTransaction** y se finaliza con el método **Commit** para confirmar las operaciones y con el método **RollBack** para cancelar las operaciones.

Ejemplo:

```
SqlConnection mCon = new SqlConnection(mStrCon);
mCon.Open();
SqlTransaction mTr = mCon.BeginTransaction();
Try {
      SqlCommand mCom = new SqlCommand();
      mCom.Connection = mCon;
      mCom.Transaction = mTr;
      string mStrCommandText = "UPDATE …String para restar los $100"
      mCom.CommandText = mStrCommandText;
      mCom.ExecuteNonQuery();
      string mStrCommandText = "UPDATE…String para sumar los $100"
      mCom.CommandText = mStrCommandText;
      mCom.ExecuteNonQuery();
      mTr.Commit();}
catch{
      mTr.RollBack();}
finally{
      mCon.Close();}
```

Las transacciones pueden tener diferentes niveles de aislamiento, es decir, diferentes maneras de comportarse frente a operaciones concurrentes. Los niveles de aislamiento solo se pueden especificar en el constructor de la transacción:

Ej:

```
mCon.BeginTransaction(IsolationLevel.ReadCommitted);
```

Stored Procedures: ADO.NET soporta la ejecución de procedimientos almacenados parametrizados o no. Basta con especificar el CommandType del objeto Command y el nombre del procedimiento.

Ej:

```
mCom.CommandType = CommandType.StoredProcedure;
mCom.CommandText = "storedp_PersonaListar";
SqlDataReader mDr = mCom.ExecuteReader();
```

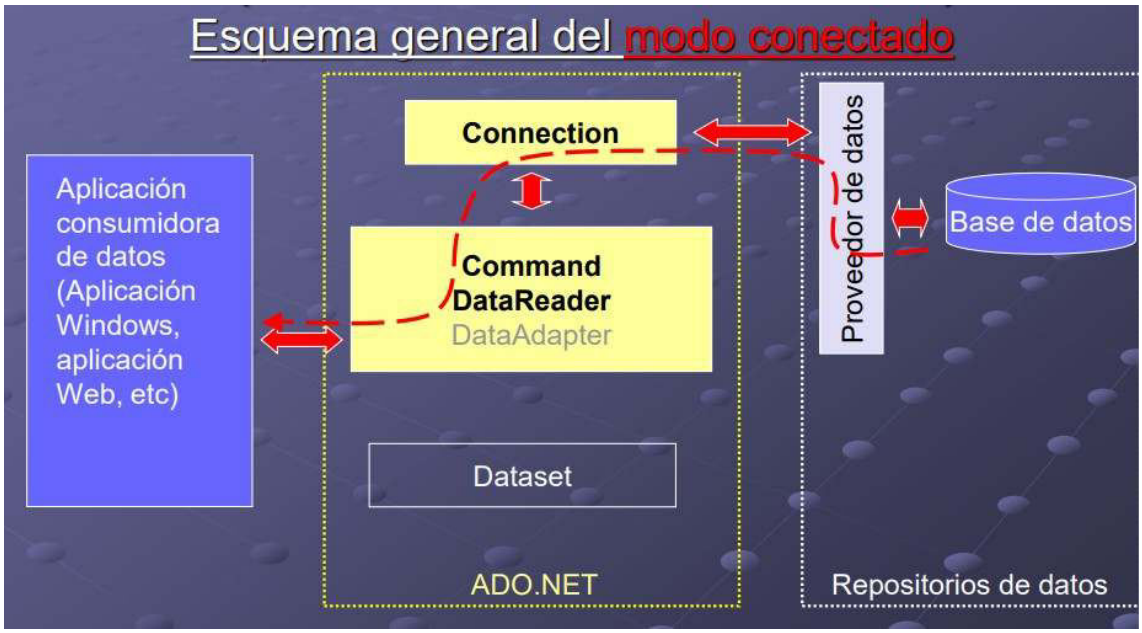
Para acompañar parámetros en la ejecución de procedimientos se utiliza la clase Parameter (dependiente del Provider).

```
Ej: mCom.CommandType = CommandType.StoredProcedure;
    mCom.CommandText = "storedp_PersonaActualizar";
    SqlParameter mPar = new SqlParameter("@Persona_nombre",
SqlDbType.VarChar);
    mPar.Value = "Juan";
    mCom.Parameters.Add(mPar);
    mCom.ExecuteNonQuery();
```

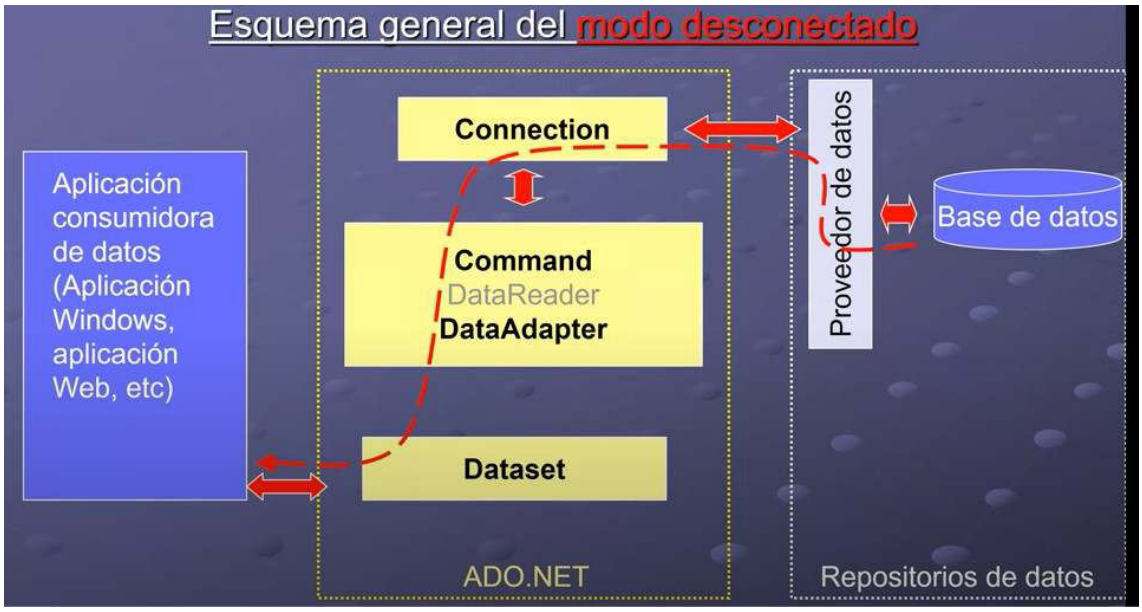
Si no se quiere incorporar cada parámetro de forma manual, se puede invocar el método compartido DeriveParameters() de la clase SqlCommanBuilder:

```
Ej: mCom.CommandType = CommandType.StoredProcedure;
    mCom.CommandText = "storedp_PersonaActualizar";
    SqlCommandBuilder.DeriveParameters(mCom);
    mCom.Parameters("@Persona_nombre").Value = "Juan";
    mCom.ExecuteNonQuery();
```

SQL Injection: usar parámetros para que no me metan por ej. 'OR 1 == 1'; Y CON ESO entran a la base de datos.



Clase 2 – Modo Desconectado: minimiza el tiempo de conexión.



En vez de un DataReader se usa un DataAdapter.

Ventajas del modo desconectado:

- Es mas escalable porque consume mas recursos del cliente que del servidor (agregar mas clientes no impacta en el mismo)
- No produce bloqueos a nivel base de datos (excepto cuando se recupera o persiste información).

Desventajas:

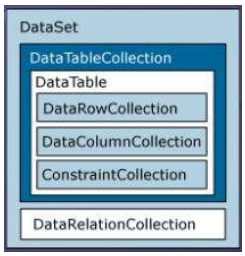
- Consume algunos recursos mas a nivel memoria en el cliente
- Es mas complejos para entornos sin estado (ej. ASP.NET)

1. OBJETO DATASET: este objeto permite mantener en memoria una porción de los datos de la base de datos para leerlos, modificarlos y organizarlos. Y luego estos se pueden persistir en la BD.

Para mantener los datos en memoria el **DataSet** replica un esquema similar al de la base de datos, con un conjunto de tablas (**DataTables**) interrelacionadas entre si (**DataRelation**), y todas ellas con columnas (**DataColumns**) y filas (**DataRows**).

Este objeto se encuentra en el espacio de nombre: **System.Data.** Por lo tanto, NO ES dependiente del proveedor de datos. Es uno general, sin importar que se use un SqlConnection o OracleConnection, etc. El DataSet será igual.

Existe una relación jerárquica dentro de los DataSet:



Principales atributos:

- DataSetName: nombre de la instancia.
- Tables: devuelve la colección de objetos Data Tables.
- Relations: devuelve la colección de objeto DataRelatios.
- DefaultViewManager: permite filtrar una vista de los datos.

Principales métodos:

- AcceptChanges: acepta los cambios en memoria.
- RejectChanges: descarta los cambios pendientes.
- Merge: combina el dataset con otro o con un DataTable o una matriz de DataRows.
- Clone: duplica el esquema en un nuevo DataSet.
- Copy: duplica el esquema y los datos en un nuevo DataSet.
- GetChanges: obtiene un DataSet con las filas con cambios pendientes.

2. OBJETO DATATABLE: representa una tabla en una base de datos. Posee filas (datarows), columnas(datacolumns), índices y claves (Constraints) y relaciones (DataRelation).

Principales atributos:

- TableName: nombre de la tabla.
- DataSet: devuelve el dataset al que pertenece. Puntero al dataset.
- Rows y Columns: devuelve una colección de objetos datarows y datacolumns.
- ChildRelations y ParentRelations: devuelve una colección de datarelation de las cuales posee la ForeignKey o la clave de origen.
- Constraints: devuelve la colección de Constraint.
- PrimaryKey: devuelve un array de DataColumn que representa la clave primaria. Es un Array porque puede haber llaves compuestas. (NOTA: PROFESOR PREFIERE LLAVES TECNICAS claves que no tienen relación con el dominio del problema, del negocio. Si no, que son agregadas).
- DefautlView: devuelve el objeto DataView.

Métodos:

- AcceptChanges, RejectChanges, Reset, Clone, Copy, Clear, GetChanges: igual que con el DataSet.
- NewRow: devuelve una nueva fila para el datatable.
- ImportRow: importa un DataRow manteniendo sus datos, cambios, estado y errores.
- Select: devuelve una matriz de DataRow resultante de una expresión de filtrado.

Eventos:

- ColumnChanging, RowChanging, RowDeleting: se lanzan cuando una columna o fila están siendo cambiadas y cuando una fila esta siendo borrada.
- ColumnChanged, RowChanged, RowDeleted: similar al anterior, pero se lanza cuando la acción de cambio o borrado ha finalizado.

3. OBJETO DATAROW: representa una fila en un DataTable. No TIENE CONSTRUCTOR PUBLICO. Por lo cual no se puede instanciar directamente. Para crear uno nuevo es a través del método NewRow.

```
DataRow mDr = mDataTable.NewRow();
```

Principales atributos:

- Item: permite acceder a la colección de DataColumns
- RowState: indica el estado de la fila (unchanged, modified, added, deleted o detached).

Métodos:

- AcceptChanges, RejectChanges: acepta o descarta cambios del DataRow.
- Delete: elimina la fila.
- GetChildRows: devuelve las filas hijas. (si hay DataRelation).
- GetParentRow/s: devuelve la/las fila/s padre según una DataRelation establecida.

4. OBJETO DATACOLUMN: representa una columna en un DataTable.

Principales atributos:

- ColumnName: nombre de la columna.
- DataType: tipo de valor de la columna.
- AllowDBNull: indica si acepta valores nullos.
- Unique: indica si la columna permitirá valores duplicados.
- DefaultValue: valor por defecto.
- Expression: expresión utilizada para columnas calculadas.
- AutoIncremente, AutoIncrementSeed, AutoincrementStep: valores para determinar si la columna será autonumérica y la forma en que comportara el incremento.
- Table: tabla a la que pertenece el datacolumn.
- Ordinal: numero de orden de la columna en la colección de DataColumCollection.

5. OBJETO DATAVIEW: representa una vista de un DataTable. Suele ser el resultado de una consulta, filtro o reordenamiento de los datos. Se puede agregar, eliminar y actualizar datos como un DataTable.

Ejemplos: Armando un DataSet

```
DataSet mDs = new DataSet("MiDataSet");
DataTable mDt = new DataTable("Persona");

DataColumn mDc = new DataColumn("Persona_Id", typeof(int));

mDt.Columns.Add(mDc);
mDt.Columns.Add("Persona_Nombre", typeof(string));
mDt.Columns.Add("Persona_Apellido", typeof(string));
mDt.Columns.Add("Persona_Direccion", typeof(string));
mDt.Columns.Add("Persona_Pais_Id", typeof(int));

mDs.Tables.Add(mDt);
```

mDs.Tables.Add(mDt); se está agregando el datatable al dataset. Pero este método tiene sobrecarga, se puede hacer así

mDs.Tables.Add("país"); se esta agregando al dataset una tabla nueva, vacía que se va a llamar "País".

Otra forma de agregar una tabla, y ponerle las columnas:

```
mDs.Tables.Add("Pais");

mDs.Tables["Pais"].Columns.Add("Pais_Id", typeof(int));
mDs.Tables["Pais"].Columns.Add("Pais_Nombre", typeof(string));

mDs.Tables["Pais"].PrimaryKey = new
        DataColumn[]{mDs.Tables[1].Columns["Pais_Id"]};

mDs.Relations.Add("Persona_Pais_Id_Pais_Id",
        mDs.Tables[1].Columns["Pais_Id"],
        mDs.Tables[0].Columns["Persona_Pais_Id"]);
```

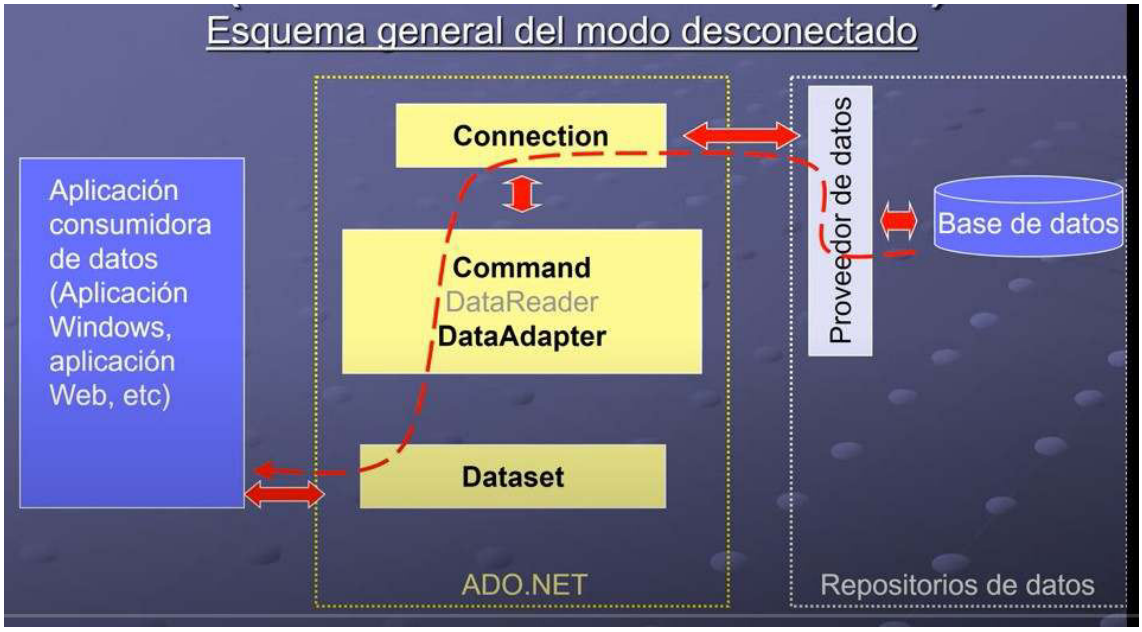
Además de con el nombre, se puede poner índice. mDs.Tables["país"] o mDs.Tables[1], el índice. 1 es País, el 0 es el anterior data table de persona.

Ejemplos: Llenando un DataSet

```
mDs.Tables["Pais"].Rows.Add(1, "Argentina");
mDs.Tables["Pais"].Rows.Add(2, "Chile");
mDs.Tables["Pais"].Rows.Add(3, "Brasil");

mDs.Tables["Persona"].Rows.Add(1, "Jose", "San Martín", "Libertador
1212", 1);
mDs.Tables["Persona"].Rows.Add(2, "Bernardino", "Rivadavia", "Rivadavia
2150", 1);
mDs.Tables["Persona"].Rows.Add(3, "Bernardo", "O Higgins", "O Higgins
710", 2);
```

Clase 3 – Modo desconectado parte II:

Objeto DataAdapter:



1. OBJETO DataAdapter: este objeto es un nexo entre un DataSet y una fuente de datos externa (base de datos, archivo, etc).

Este objeto permite rellenar uno o mas objetos DataTables con datos y nuevamente persistirlos, luego de ser manipulados de forma totalmente desconectada.

Es **DEPENDIENTE** del proveedor, ya que debe conocer como leer y actualizar el repositorio. Ej. Para SQL: SqlDataAdapter. Oracle: OracleDataAdapter, etc.

Principales atributos:

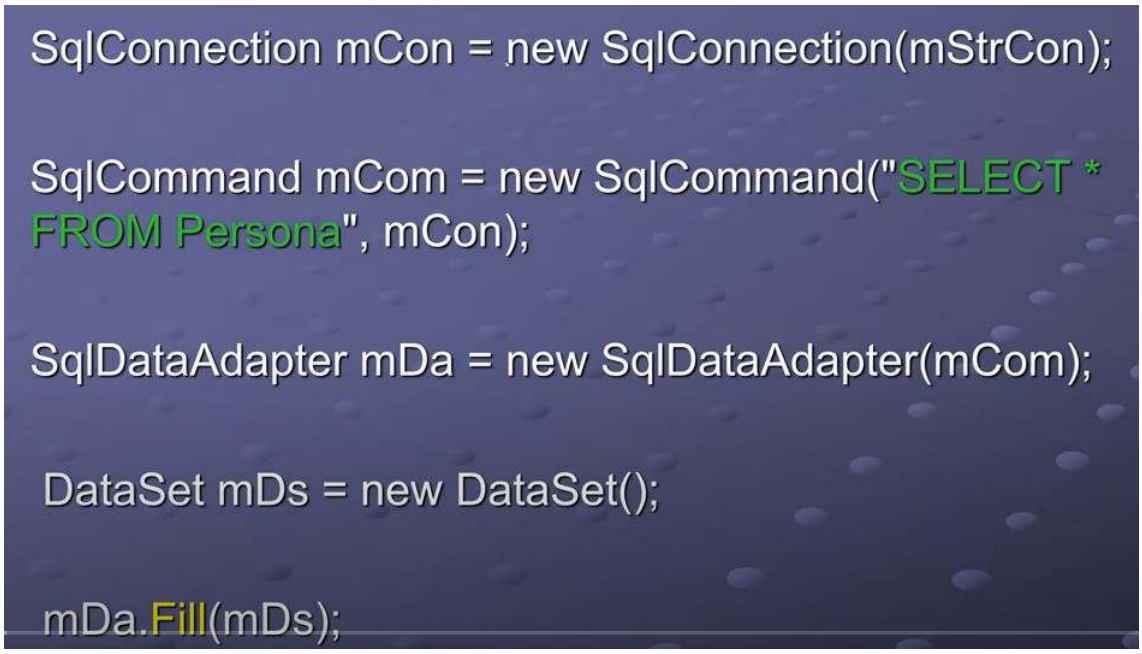
- SelectCommand, UpdateCommand, InsertCommand, DeleteCommand: indican la instrucción SQL a ser utilizada. *(son objetos Command).*
- TableMappings: permite asociar correspondencia entre columnas y tablas del DataSet y el origen de datos.

Principales métodos:

- Fill: agrega o actualiza DataRows en el DataSet con datos del origen. Si el DataSet no tiene creado un DataTable, crea uno con la estructura del resultado que arrojo.
- FillSchema: agrega una tabla en un DataSet con el mismo esquema de la tabla de origen.
- Update: actualiza el origen de datos con los cambios efectuados en el DataSet.

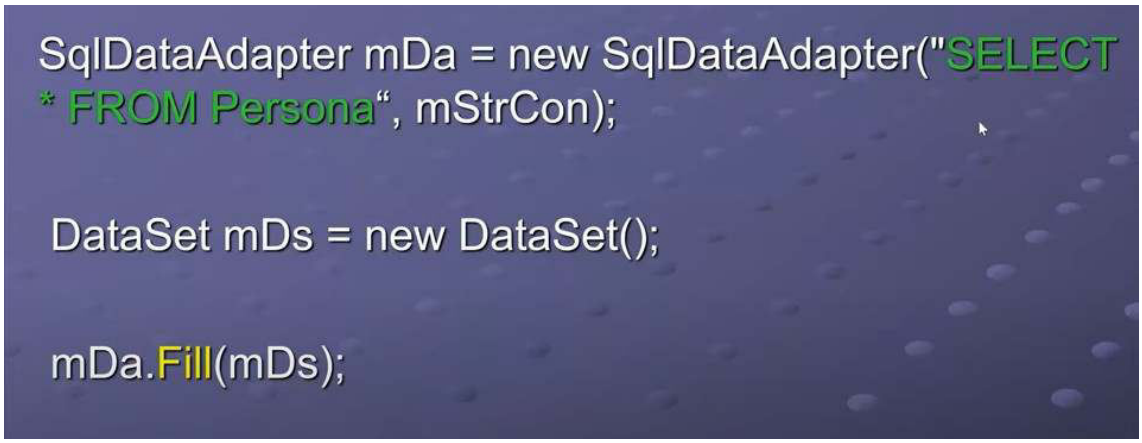
Eventos:

- RowUpdating: se lanza antes de enviar un comando SQL que actualice el origen de datos.
- RowUpdated: se lanza después de enviar un comando SQL que actualice el origen de datos.
- FillError: se lanza cuando ocurre un error durante la operación Fill.



El string del SqlCommand, es el commandText.

El dataAdapter sume que es un select command. Si se le pasa en el constructor asume que es un selectCommand.



El SqlDataAdapter recibe 2 parámetros en el constructor: en este caso es primero un string, y 2do el connectionString:

- Primero instancia un SQl connection con el connection string.
- 2do instancia un SqlCommand y le asigna el commandtext, el primer string.
- Al SqlCommand le asocia la conexión que creo y después ese SqlCommand se lo asigna al selectCommand del DataAdapter.

```
SqlConnection mCon = new SqlConnection(mStrCon);

mCon.Open();

SqlDataAdapter mDa = new SqlDataAdapter("SELECT * FROM
Persona", mCon);

 DataSet mDs = new DataSet();

 mDa.Fill(mDs, “NombreTabla”);

mCon.Close();
```

Se le agrega al final, el nombre de la tabla. De como quiero que se llame. Cuando hace el Fill.

Fill chequea como está la conexión, si ya se abrió no va a tocar nada el Fill, si no está abierta la abre, pero si esta abierta no hace nada y después se debe cerrar manual.

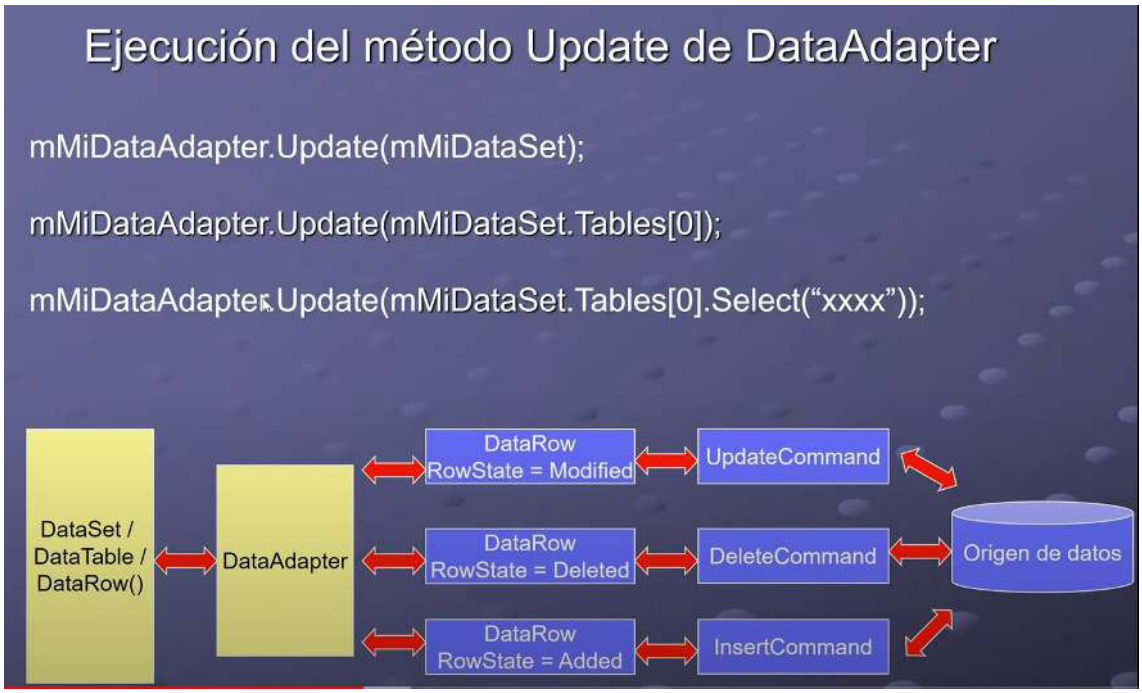
Mapeo de columnas: por defecto DataAdapter le da el nombre a las columnas del Data Table iguales a los nombres de la tabla del origen de datos. Si necesitamos nombres diferentes hay que efectuar un mapeo mediante objetos *DataTableMapping*

```
SqlDataAdapter mDa = new SqlDataAdapter("SELECT persona_Id,
persona_nombre FROM Persona", mStrCon);

DataSet mDs = new DataSet();
mDs.Tables.Add(“MiTabla”);
mDs.Tables[0].Columns.Add("Id", typeof(int));
mDs.Tables[0].Columns.Add(“Nombre", typeof(string));

mDa.TableMappings.Add(“TablaDB”, “MiTabla”);
mDa.TableMappings[0].ColumnMappings.Add (“persona_id”,”Id”);
mDa.TableMappings[0].ColumnMappings.Add (“persona_nombre”,”Nombre”);

 mDa.Fill(mDs, “TablaDB”);
```

TablaDB en realidad seria Persona. Y al final en el fill, debería ir “MiTabla”

```
SqlDataAdapter mDa = new SqlDataAdapter("SELECT persona_Id,
persona_nombre FROM Persona", mStrCon);

DataSet mDs = new DataSet();
mDs.Tables.Add("MiTabla");
mDa.TableMappings.Add("TablaDB", "MiTabla");
mDa.TableMappings[0].ColumnMappings("persona_id","Id");
mDa.TableMappings[0].ColumnMappings("persona_nombre","Nombre");

mDa.FillSchema(mDs, SchemaType.Mapped, "TablaDB");

mDa.Fill(mDs, "TablaDB");
```

ACTUALIZACION DE DATOS: DataAdapter permite grabar los cambios realizados en el DataSet en el origen de datos. El método Update() ejecuta un batch de operaciones contra la base de datos dependiendo del estado de cada fila ej. Si el estado de la fila es "agregada", va a ejecutar un INSERT, si el estado es "eliminada" ejecuta un DELETE, etc.



El Data Adapter puede tener de parámetro un dataset, un data table o un array de datarows o un update sobre ciertas filas. Lo que se hace al momento de persistir los datos, es iterar por cada registro y revisar su estado y según este ejecuta la instrucción INSERT, DELETE, UPDATE, ETC. (Si se agrego una fila al DataSet y luego se quita, ese estado es QUITADO, es una fila que nunca existió en la BD, por lo tanto, no hay que realizar ninguna operación contra la BD.

Preparación del objeto DataAdapter: se le debe especificar los métodos específicos del proveedor para actualizar, eliminar e insertar datos en la BD. Una forma de generarlos automáticamente es con el objeto CommandBuilder:

1. Se instancia un DataAdapter

```
SqlDataAdapter mDa = new SqlDataAdapter("SELECT * FROM
Persona", mConStr);
```

2. Se instancia un SqlCommanBuilder y se le pasa como parametro el DataAdapter:

```
SqlCommandBuilder mCb = new SqlCommandBuilder(mDa);
```

3. Al DataAdapter se le prepara cada método (update, delete, insert) con el CommanBuilder, se le asocia un objeto Command preparado para realizar la instrucción.

```
mDa.UpdateCommand = mCb.GetUpdateCommand();
mDa.DeleteCommand = mCb.GetDeleteCommand();
mDa.InsertCommand = mCb.GetInsertCommand();
```

Este método GetXXX genera el código SQL necesario para efectuar la instrucción en la base especificada en la conexión del DataAdapter.

El código SQL resultante de esta operación:

```
mDa.DeleteCommand = mCb.GetDeleteCommand();
```

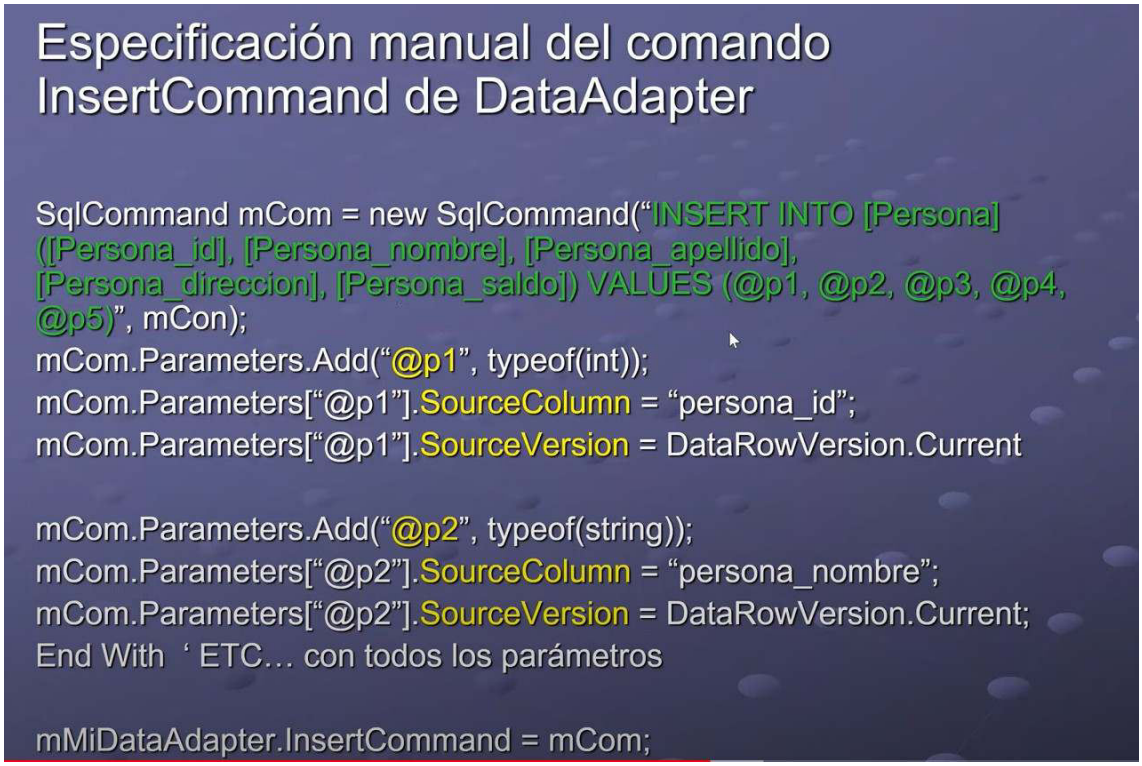
Será:

```
"DELETE FROM [Persona]
WHERE (([Persona_id] = @p1) AND
((@p2 = 1 AND [Persona_nombre] IS NULL) OR
([Persona_nombre] = @p3)) AND ((@p4 = 1 AND
[Persona_apellido] IS NULL) OR
([Persona_apellido] = @p5)) AND
((@p6 = 1 AND [Persona_direccion] IS NULL) OR
([Persona_direccion] = @p7)) AND ((@p8 = 1 AND
[Persona_saldo] IS NULL) OR
([Persona_saldo] = @p9)))"
```

El DeleteCommand generaría algo así. El delete bastaría con saber la primary key, pero le mete todos los valores restantes para hacer un control de concurrencia. Controla todos los campos, va a borrar tal registro cuando tal campo tenia X valor, y así pregunta en cada uno. ¿Pero qué pasa si estos valores cambiaron? No se va a realizar ningún delete. Va a devolver 0 filas

afectadas. Se debe recordar el valor anterior cuando se consultó, y el actual para hacer bien el control de concurrencia.

Se puede especificar manualmente cada uno de los comandos de actualización agregando los parametros necesarios pero se debe especificar que valor se le asignara y que versión de la información del DataRow tomar (actual == Current: lo que tengo en memoria con los cambios o anterior == original: lo último que leí de la BD).



Manejo de errores:

Pueden ocurrir errores al actualizar los datos, ej. Errores de concurrencia (otro usuario modifico una fila antes de actualizar). Se le puede indicar al DataAdapter que actitud tomar ante un error. Las opciones son:

- Continuar con el resto de los cambios y dejar la fila con el estado actual (added, deleted, modified, etc).
- Detener el proceso y lanzar una excepción. Con try, catch.

En el caso que se quiera continuar con el resto de los cambios, se puede verificar luego las filas conflictivas.

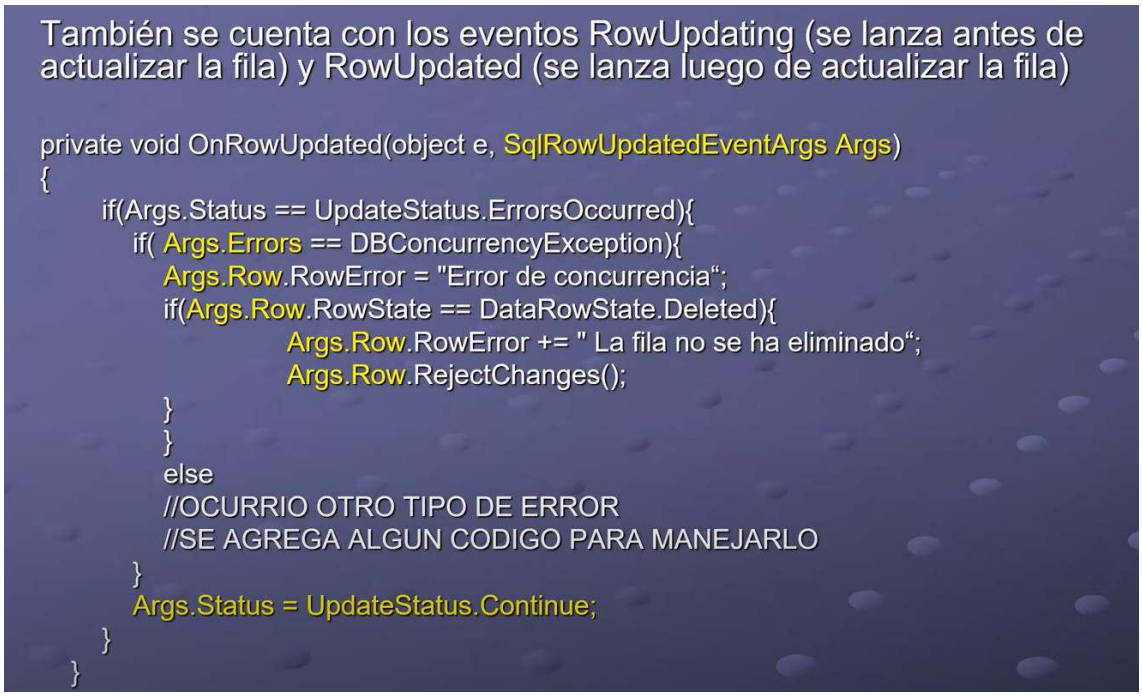
```
mMiDataAdapter.ContinueUpdateOnError = true;

mDa.Update(mDs.Tables[0]);

if (mDs.Tables[0].HasChanges)
{
        MessageBox.Show("Quedan aún sin actualizar " +
        mDs.Tables[0].GetChanges().Rows.Count + " filas");

}
```

Luego puede agregarse código para manejar los errores

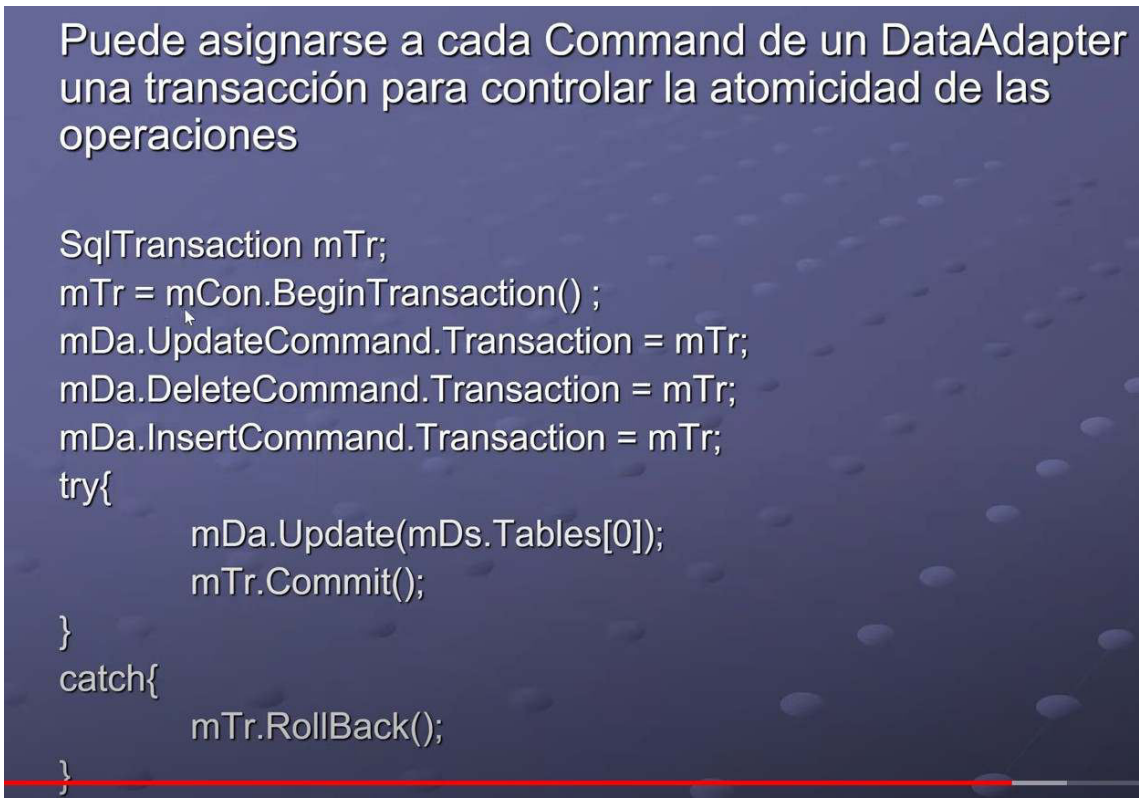
```
DataTable mDt;
mDt = mDs.Tables[0].GetChanges();

foreach(DataRow mDrow In mDt.Rows)
{       switch(mDrow.RowState) {
                case  DataRowState.Modified:
                        //CODIGO PARA MANEJAR EL ERROR
                        break;
                case  DataRowState.Added:
                        //CODIGO PARA MANEJAR EL ERROR
                        break;
                ETC…}
}
```

GetChanges: devuelve un datatable con los cambios pendientes.

Y con eso para cada fila, se hace un switch para saber que se quiso hacer y fallo.



Si lo que fallo fue un borrado, puedo hace run RejectChanges este devolvía al estado anterior, lo saca de deleted al unchanged. Así por ej. En una grilla al volver al estado anterior, se volvería a mostrar a la grilla.



Para usar ConfigurationManager necesito: esto para poder utilizar el connectionString en el AppConfig.

- 1 Using System.Configuration;

- 2 agregar la referencia a System.Configuration

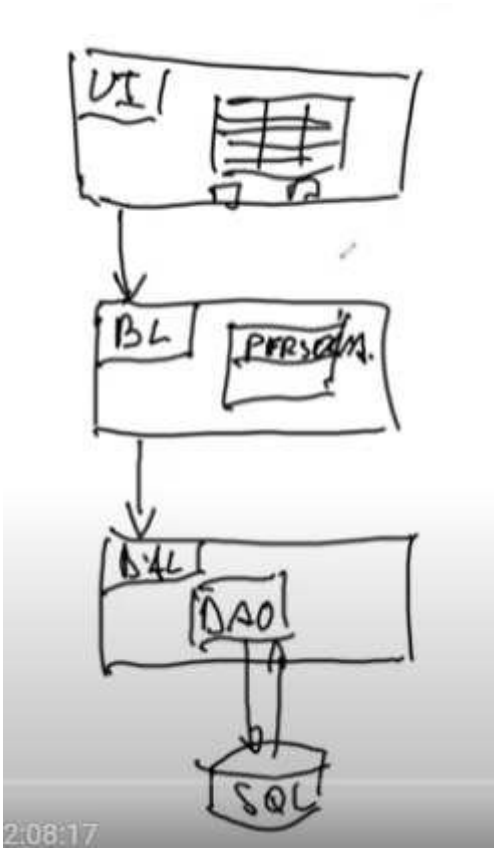
Clase 4: Persistencia en capas

Se va a utilizar el modo conectado y desconectado según que nos convenga más.

Dividir una aplicación en capas:

1. Interfaz de usuario: UI – su responsabilidad es mostrar a los usuarios cosas y permitir la interacción.
2. Business layer: BL – en esta capa van a estar las clases del negocio. Ej. Clase persona, compras, factura, etc.
3. Data Acces Layer: DAL – Acá van a estar los objetos de accesos de datos.
4. Business Entity: BE – van a estar todas las clases del negocio, pero solamente los atributos.

Mas abajo, ya no como una capa va a estar la base de datos.



Desarrollar así sirve para lograr independencia. Ej. Se puede Re implementar la interfaz a web y todo lo de abajo se mantiene igual.

Importante que la separación de responsabilidades sea clara y se respete.

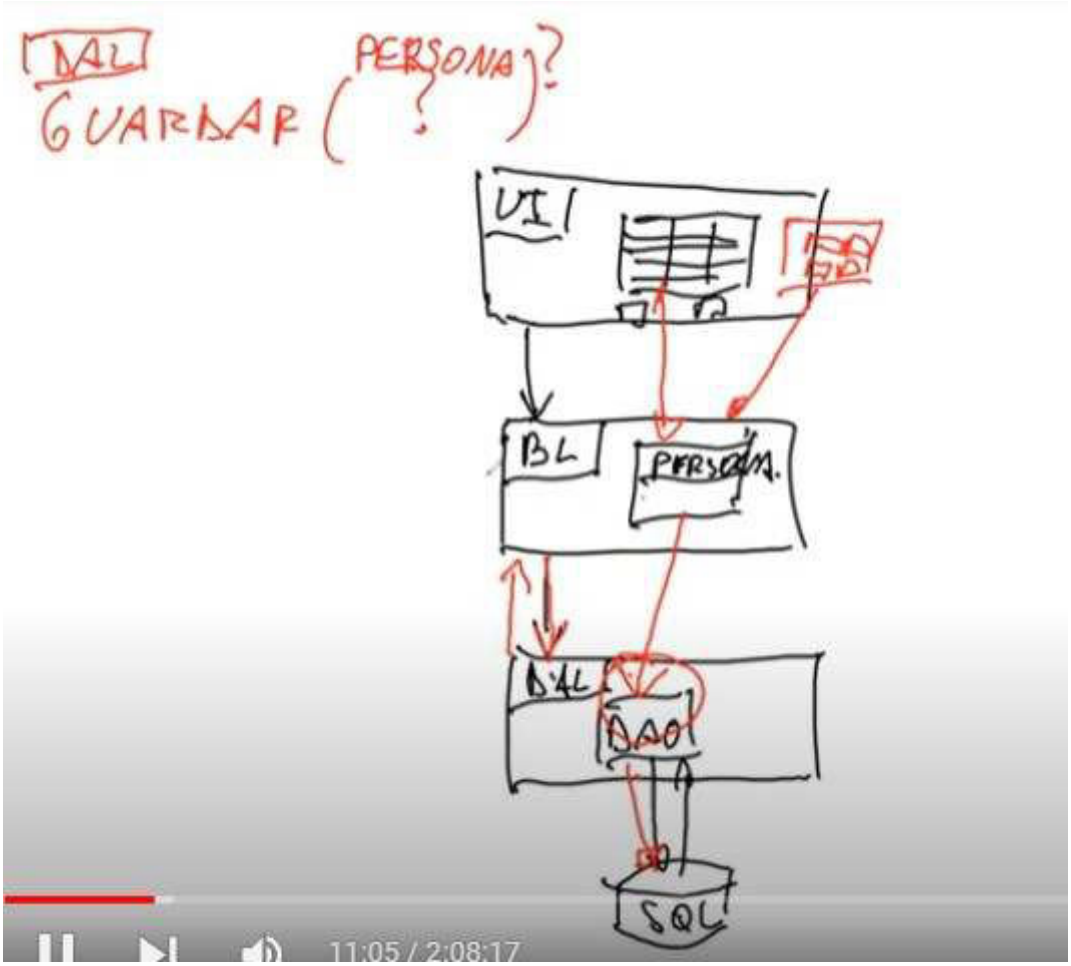
Si la interfaz de usuario es mostrar interfaz, no vamos a mostrar otra cosa.

Todo negocio, todas las clases van a ir a la BL con sus reglas. Y cualquier cambio va a ser en esa capa solamente.

Si después se dice que se va a persistir en Oracle en vez de sql, se re implementa la capa DAL, pero las capas anteriores no se debe tocar nada.

Interfaz de usuario va a pedir cosas, como listar personas a la BL, y la BL va a pedir a la DAL, persistir la información. Pero hay un problema:

Si estamos persistiendo una persona con un método guardad y esperamos como parámetro una persona, no se puede. Poruqe no hay referencia circular, la DAL no sabe que es persona.



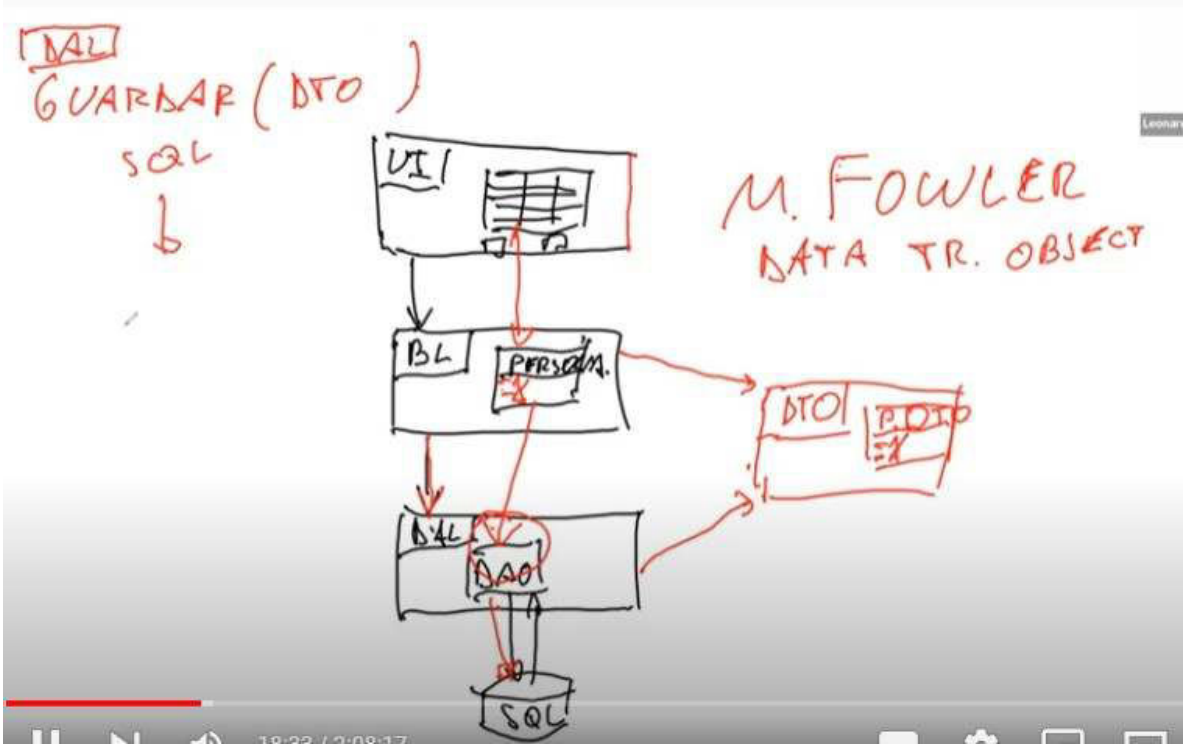
¿Como se resuelve?

1 opcion, le pasamos por parámetros los valores que nosotros queremos guardar. Método guardar(id,nombre,apellido). Pero si una clase tiene muchos atributos no es muy eficiente, no es OPCION ESTA.

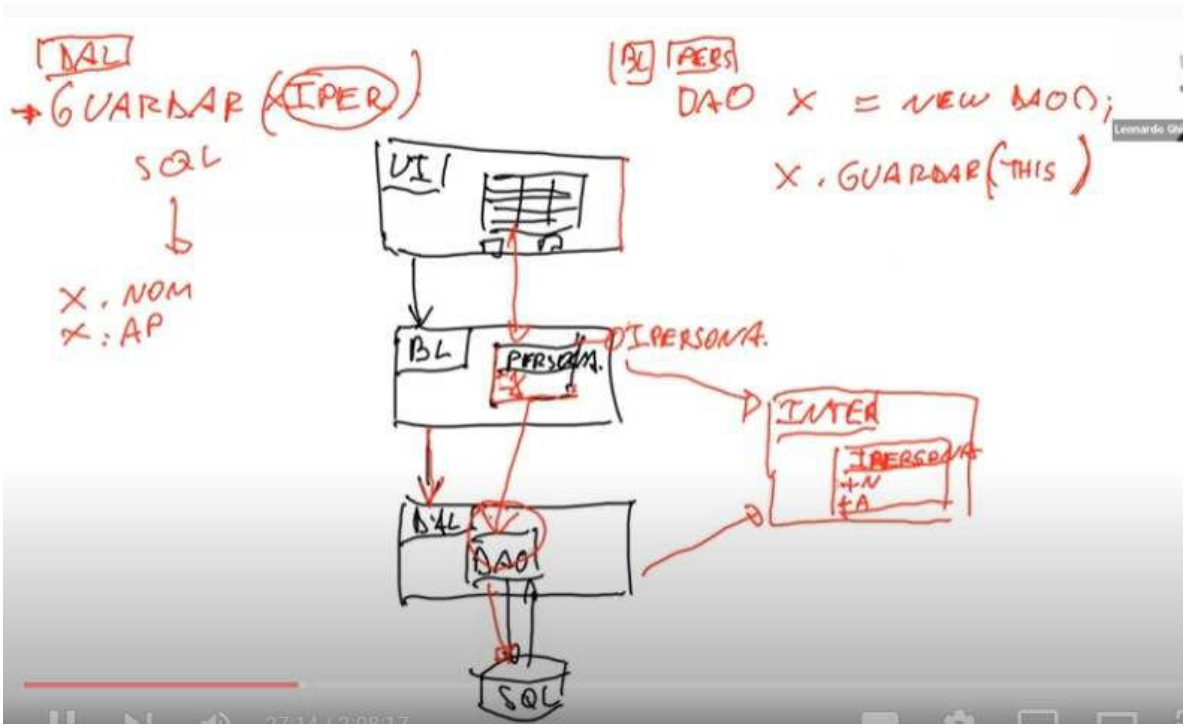
Hay que buscar algo en común: una cosa en común es el framework.net, entonces podríamos pasarle un datarow con toda la info, pero habría que generarlo en la BL y eso es parte de la DAL. Tampoco es la mejor

MEJOR OPCION: vamos a desarrollar algo que todas las capas conozcan. Hay que buscar punto en común

1- DTO: es un patron de M.Fowler. El data tr. Object permite crear un objeto que se llame persona DOT que vaa tener todo lo que tiene persona, todos los atributos. SOLO atributos, 0 metodos. Solo va a existir a efecto de trasnferir datos. Se instancia un DTO en la BL y en la capa de datos, esperamos un DTO. Entonces pasamos todos los datos encapsulados a la DAO.



2- Proyecto de interfaces: vamos a tener un Ipersona. La interfaz es un contrato yquien la hereda se comprote a implementar los métodos y atributos. La Ipersona va a decir que toda persona va a tener un nombre y apellido, etc. Los dos proyectos van a conocer la interfaz y persona de la BL impelmenta Ipersona. Persona es una Ipersona. Y a la dal entonces le pasamos una Ipersona. Vamos a acceder a atributos de la persona sin conocer a la persona. Se usa a la interfaz para hacer este pasaje de capas. A diferencia del otro, acá le estoy pasando el mismo objeto. Le puedo pasar mi misma instancia.

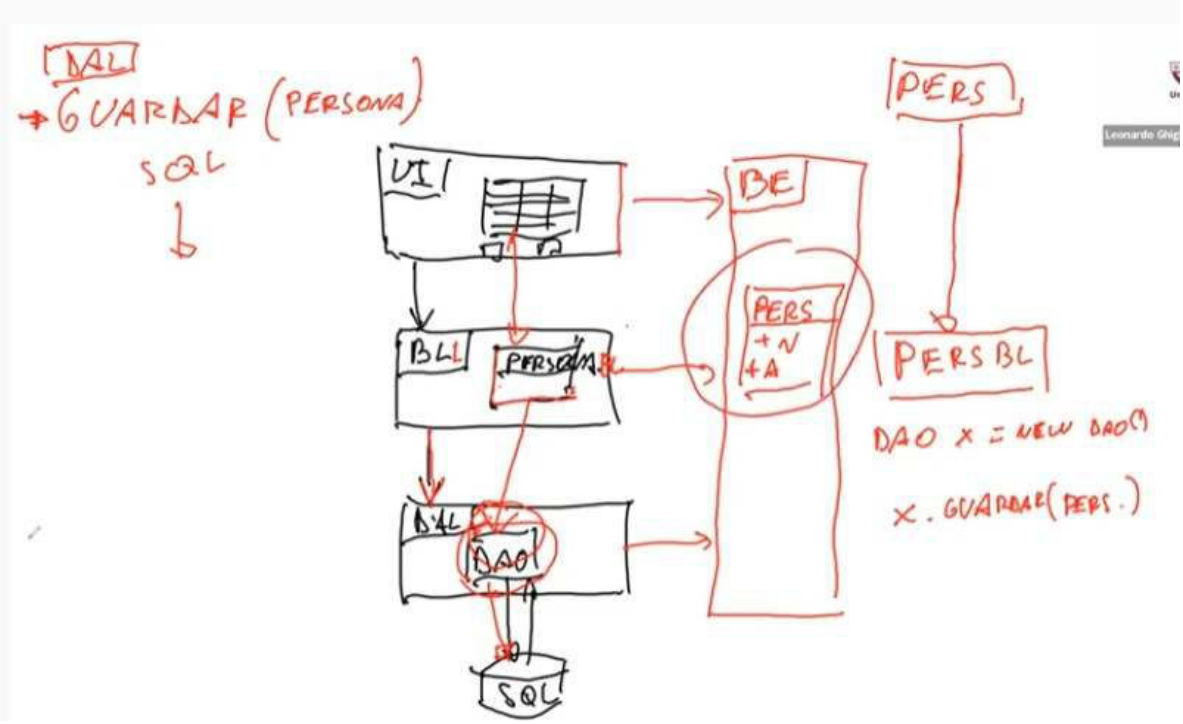


Si hacemos crecer el DTO, que no conozca solo los últimos dos, si no que sea conocido por todos, por todas las capas.

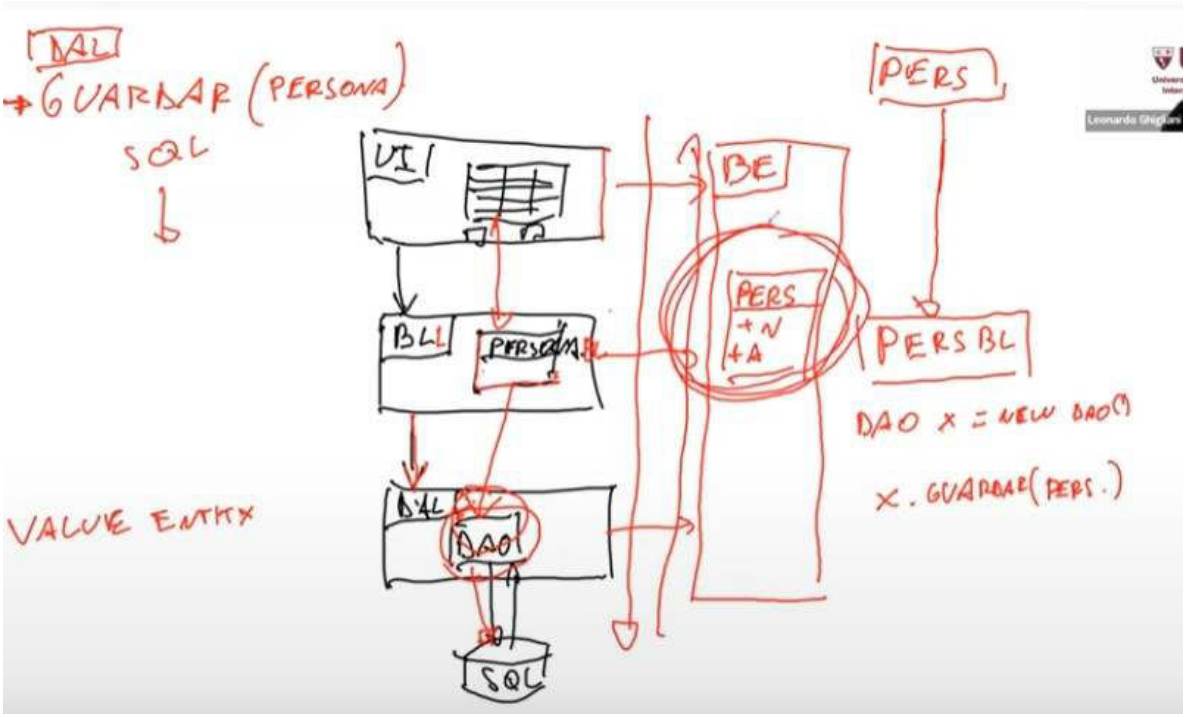
Se va a llamar BE bisnis entity. Y este va a tener un objeto que representa el estado de una persona. Nombre, apellido, etc.

La consecuencia es que, esto va a ser capa lógica, y en la BL, persona va a ser Persona BL. Bl va a tener todos los métodos y la BE va a tener todos los atributos. Entonces voy a tener separada la entidad.

Si desde la ui necesito guardar info, voy a instanciar una BE y le voy a pasar todos los atributos, PersonaBl v a instanciar una DAO, y le vamos a pasar una persona.



Entonces el objeto persona de la BE, llamado también value entity, es una entidad solo de valor es transversal a todas las capas:



Este objeto va a ser el encargado de tener toda la información.

Ventajas: es bastante liviano solo tiene atributos, es mas eficiente.

Separado atributos y mantenimiento, mantenimiento se simplifica.

Desventajas: la entidad dele dominio se divide.

DAO: tiene un problema la DAL. Se persiste una persona, pero si en el dominio tengo varias clases: persona, cliente, producto, venta, etc.

La dal va tener que tener un método guardar para cada clase. Va a haber muchas líneas de código. Se va a tener mucho acoplamiento así.

Entonces se va a hacer clases especificas para persisitr a cada una de estas.

Todo lo clásico de ado.net va a ir a la DAO, y se va a gregar una clase a la DAL, en este caso ej. PersonaDAL (va a ser la clase que sepa como guardar una persona) entonces se desacopla, DAO solo se conecta y guarda pero como guardarla lo define PersonaDAL y así cada clase según las clases que necesite el dominio.

ENRN entidad de neogico, regla de negocio.

BE Clase con solos atributos.

BL Tiene solo métodos ej. Obtener persona, listar persona, guardar persona, eliminar persona, etc. BL no conoce el método de persistencia, solo le pide a la DAL hacer tal cosa.

TODA REGLA DE NEGOCIO debería ir acá, en la BL. Pero ahora es un simple ABM.

```
        if (mDs.Tables.Count > 0 && mDs.Tables[0].Rows.Count > 0)
        {
            foreach (DataRow mDr in mDs.Tables[0].Rows)
            {
                Persona mPersona = new Persona(int.Parse(mDr["Persona_Id"].ToString()));
                ValorizarEntidad(mPersona, mDr);
                mPersonas.Add(mPersona);
            }
        }
        return mPersonas;
    }

    2 referencias
    private static void ValorizarEntidad(Persona pPersona, DataRow pDataRow)
    {
        pPersona.Nombre = pDataRow["Persona_Nombre"].ToString();
        pPersona.Apellido  = pDataRow["Persona_Apellido"].ToString();
        pPersona.Documento  = pDataRow["Persona_Documento"].ToString();
    }

}
```

En valorizarEntidad no hace falta que ponga un return, porque es un valor por referencia. Se le está pasando un puntero.

Todos los métodos son static, no hay estado en la DAL, no necesito instancia de PersonaDAL. Todos los métodos son de clases.

```
static int mId;
1 referencia
private static int ProximoId()
{
    if(mId == 0)
        mId = (new DAO()).ObtenerUltimoId ("Persona");
    mId += 1;
    return mId;
}
```

Cada instancia va a conocer el ultimoid porque es un método statico, es compartido.

OBTENER CONNECTION STRING: si bien lo uso en la DAO, el app.config va a estar en el proyecto principal, todas las DLL compilado, se van a copiar al BIN dele proyecto principal.

```
1 referencia
private void PersonaListado_Load(object sender, EventArgs e)
{
    grdPersonas.Columns.Add("Id", "Id");
    grdPersonas.Columns["Id"].Visible = false;
    grdPersonas.Columns.Add("Nombre", "Nombre");
    grdPersonas.Columns["Nombre"].Width = 200;
    grdPersonas.Columns.Add("Apellido", "Apellido");
    grdPersonas.Columns["Apellido"].Width = 200;
    grdPersonas.Columns.Add("Documento", "Documento");
    grdPersonas.Columns["Documento"].Width = grdPersonas.Width - 403;
    grdPersonas.RowHeadersVisible = false;
    grdPersonas.AllowUserToAddRows = false;
    grdPersonas.AllowUserToDeleteRows = false;
    grdPersonas.EditMode = DataGridViewEditMode.EditProgrammatically;
    grdPersonas.MultiSelect = false;
    grdPersonas.SelectionMode = DataGridViewSelectionMode.FullRowSelect;

    Actualizar();
}
```

```
iniciar proyecto WinForms
Agregar 3 Librerías de clases (BL, BE, DAL)
Referenciarlas
Empezar por las clases que no dependen de otras (Entidad BE y DAO)
Seguir Entidad DAL
Seguir por Entidad BL
Testear todo en Form Load
Implementar UI
```

Referencia del proyecto principal a System.Configuration para traer el string de conexión del app config.

```
private void Principal_Load(object sende
{

    Persona p = new Persona();
    p.Nombre = "Test";
    p.Apellido = "Prueba";
    p.Documento = "111111";
    PersonaBL pBL = new PersonaBL();
    pBL.Guardar(p);
```

Probar sin UI guardar un nuevo registro.

```
PersonaBL pBL = new PersonaBL();
Persona p = pBL.Obtener(1);
p.Nombre = "Test2";
p.Apellido = "Prueba2";
p.Documento = "22222222";
pBL.Guardar(p);
```

Modificar

```
PersonaBL pBL = new PersonaBL();
Persona p = pBL.Obtener(1);

pBL.Eliminar(p);
```

Eliminar

//**********//

Patrones de diseño: un patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema, de tal modo que se pueda aplicar esta solución un millón de veces, sin hacer lo mismo dos veces.

Son descripciones de clases y objetos relacionados que están particularizados para resolver un problema de diseño general en un contexto determinado.

The gang of four -> libro de 1995 donde se presentan los patrones de diseño. Se introdujeron 23 patrones divididos en 3 categorías:

1- **Patrones de creación**: centrados en el proceso de creación de objetos.
2- **Patrones de estructura**: enfocados en la composición estática y en las estructuras de clases y objetos
3- **Patrones de comportamiento:** enfocados en la interacción dinámica.

Elementos de un patrón:

- Nombre del patrón: describe el problema junto con su solución y consecuencias.
- Problema: problema que resuelve y el contexto. Cuando usar el patrón.
- Solución: indica el diseño de sus elementos, relaciones, responsabilidades y colaboraciones.
- Consecuencias: cuales son las ventajas y costos de aplicar el patrón
- Contexto: donde será posible aplicar el patrón.

Patrón composite

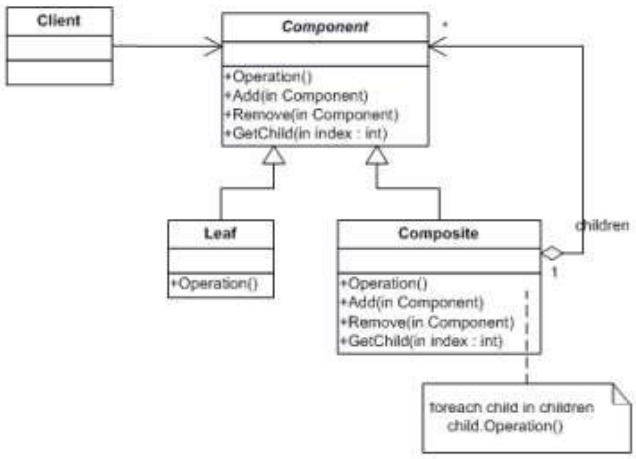
Propósito

- Compone objetos en estructuras de árbol para representar jerarquías de parte-todo.
- Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos

Aplicabilidad

- Quiera representar jerarquías de objetos todo-parte.
- Quiere que los clientes sean capaces de obviar las diferencias entre composiciones de objetos y los objetos individuales. Los clientes tratarán a todos los objetos de la estructura compuesta de manera uniforme.

- Estructura



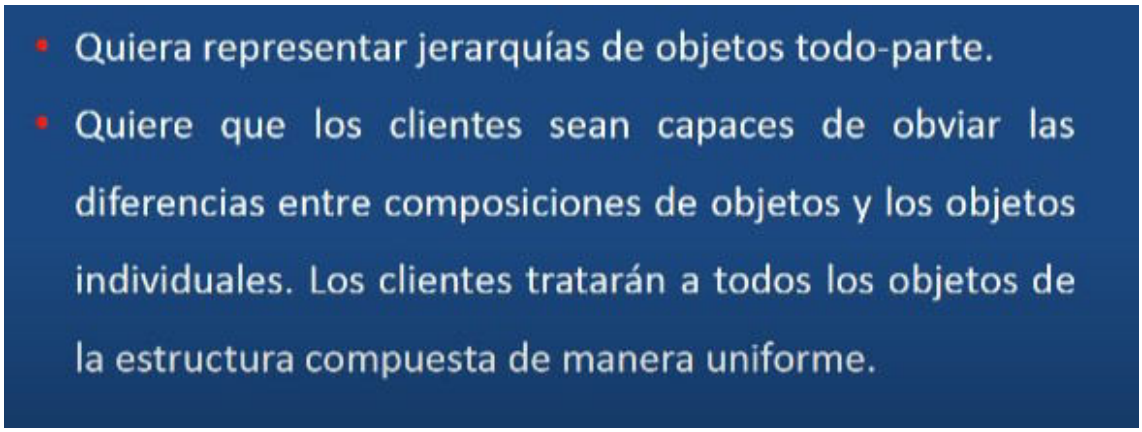
Consecuencias:

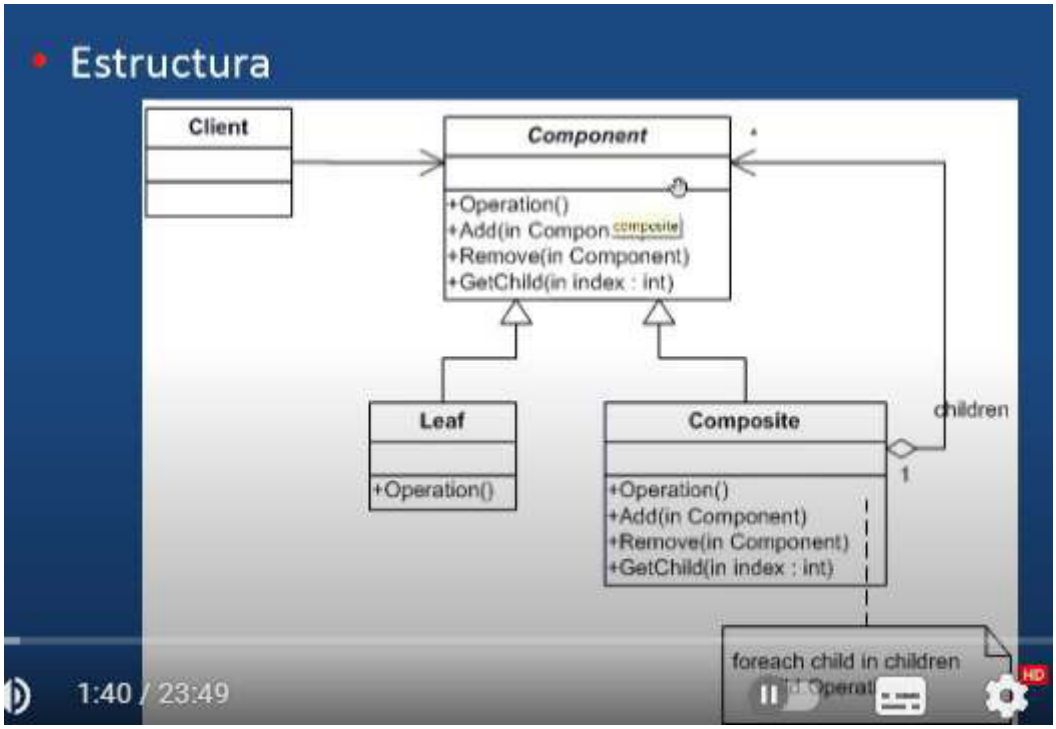
- Define jerarquías de clases formadas por objetos primitivos y compuestos.
- Simplifica al cliente. El cliente trata las estructuras uniformemente.
- Facilita añadir nuevos tipos de componentes.
- Puede hacer que un diseño sea demasiado general.

Notas:

- Crear una interfaz que oficie de "mínimo denominador común" que haga que los contenedores y los contenidos puedan ser intercambiables
- Todas las clases Contenedor y Contenido implementan la interfaz.
- Métodos de manejos de hijos (Ej. *AddChild()*, *RemoveChild()*) deberían normalmente ser definidos en la clase *Composite*. Por desgracia, el deseo de tratar a los objetos *Leaf* y *Composite* de manera uniforme requiere que dichos métodos sean movidos a la clase abstracta *Component*.
- Las clases Contenedor aprovechan el polimorfismo para delegar en sus objetos Contenido.

Composite es un patrón de diseño estructural que permite componer objetos en una estructura en forma de árbol y trabajar con ella como si fuera un objeto único.

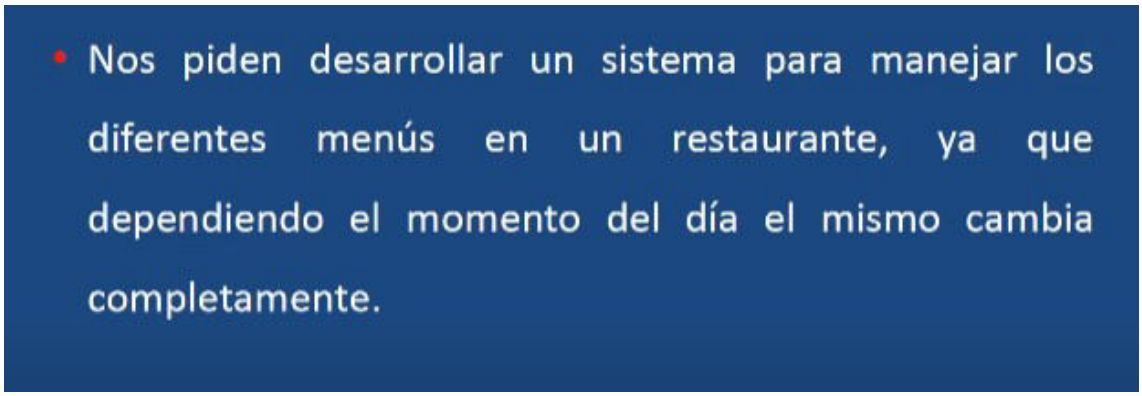




Componente, la hoja y el compuesto

Componente tiene operaciones generales, clase abstracta. Permite manipular a los hijos en la composicion. Esta se especializa en:

Una hoja y un compuesto. Componente utiliza las mismas operaciones que el componente.



Viola el principio solid, el principio de integración de interfaces.

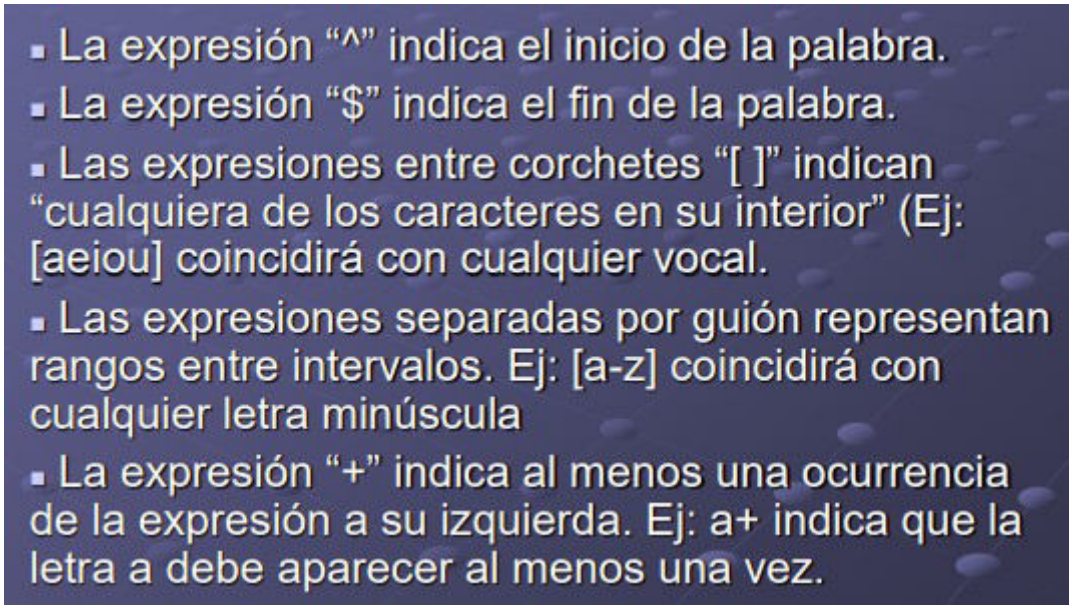
La clase hoja impliementa los métodos de manipulación de hijos.

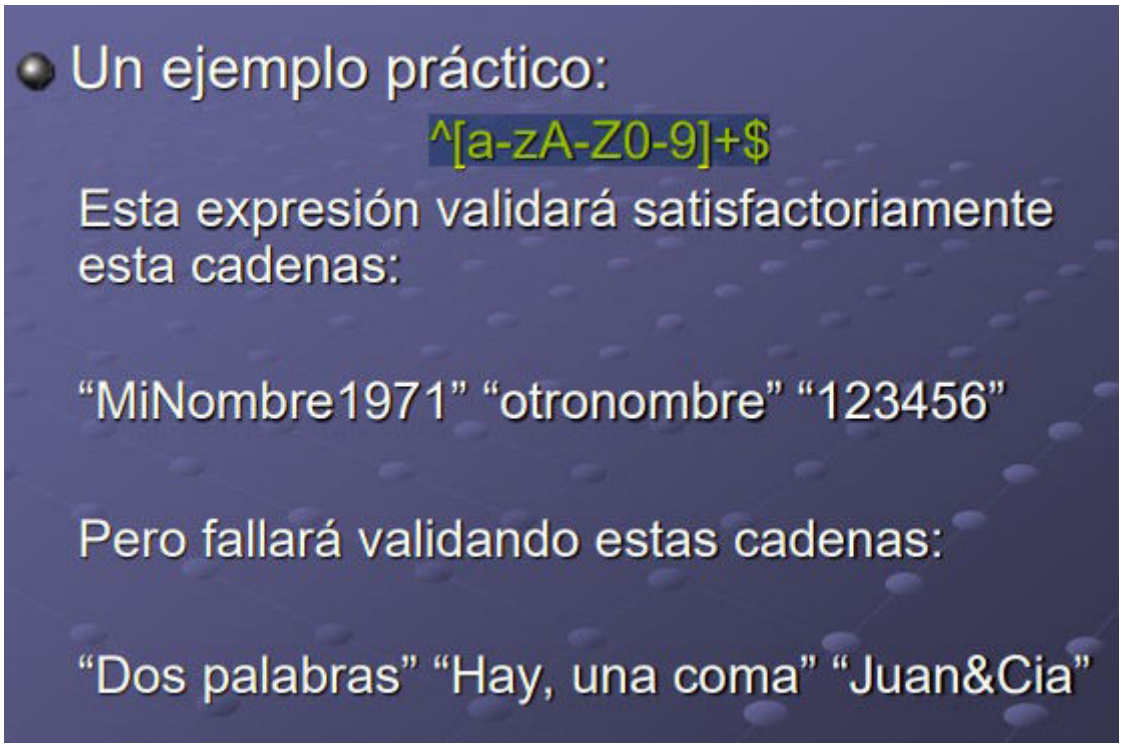
Es una estructura recursiva.

Expresiones regulares: son un mecanismo de evaluación, búsqueda y/o remplazo de cadenas de caracteres aplicando patrones en un lenguaje especifico. (patrón para evaluar cadenas de texto)

.NET provee un conjunto de clases para aplicar estas técnicas. **System.Text.RegularExpressions.**

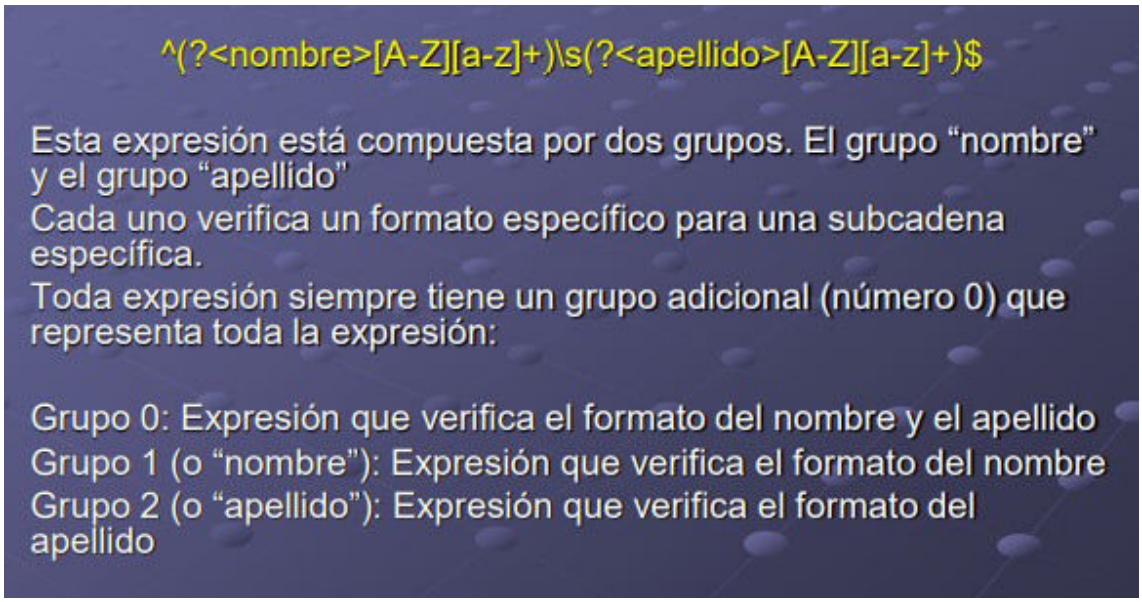
Para construirlas se debe comprender una forma de estructurar secuencias lógicas. Estas expresiones se construyen mediante la concatenación de caracteres especiales.





Expresiones regulares en grupos: algunas expresiones pueden estar formadas por grupos, que se identifican por nombre o por número.

El nombre del grupo se puede especificar con la sintaxis: **?<nombre> o ?’nombre’**



La clase Regex provista por el .NET permite manipular y aplicar expresiones regulares:

Principales métodos:

- Constructor (new): Recibe el patrón de expresión regular que usará.
- Match: Devuelve un objeto Match con la primer coincidencia del patrón en el string proporcionado como parámetro. Acepta opciones (Multiline, IgnoreCase, Compile, etc). Si no hay coincidencias devuelve un objeto Match insatisfactorio (con su atributo Success = False)
- Matches: Devuelve un objeto MatchesCollection con todas las coincidencias.
- IsMatch: Devuelve *true* o *false* indicando si el patrón encontró al menos una o ninguna coincidencia.
- Replace: Remplaza cada ocurrencia del patrón por la subcadena provista como parámetro.
- Todos estos métodos (excepto el constructor) tienen su equivalente compartido (no requieren instancia)

Manejo de grupos:

- GetGroupNames: Devuelve una matriz con los nombres de los grupos (si carecen de nombre le asigna su número como nombre).
- GetGroupNumbers: Devuelve la matriz con los números de grupos.
- GroupNameFromNumber, GroupNumberFromName: Devuelve el número de grupo del nombre especificado y viceversa.
- Groups: Da acceso a la colección de grupos.

Clase Match:

- La clase Match representa una coincidencia en la aplicación de un patrón de expresión regular.

Principales atributos:

- Success: Indica si la coincidencia fue exitosa.
- Value: Indica la subcadena coincidente encontrada.
- Index: Indica el índice de esa coincidencia en la cadena evaluada.
- Groups: Proporciona acceso a la colección de grupos de esa coincidencia.

Principales métodos:

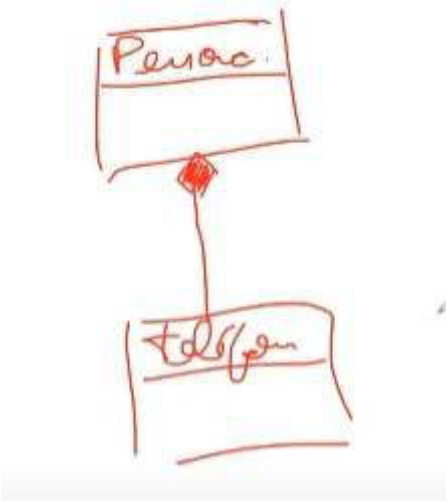
- El método NextMatch transforma la instancia de Match actual en la representación de la próxima coincidencia de la MatchesCollection.

Principales atributos:

- Value: Devuelve la subcadena coincidente con la porción del patrón de expresión regular contenido en el grupo.
- Index: Devuelve el índice de esa ocurrencia en la cadena evaluada.
- Captures: Devuelve una colección CapturesCollection que contiene todas las capturas de ese grupo que han coincidido con el patrón del grupo. El objeto Capture representa cada una de esas coincidencias.

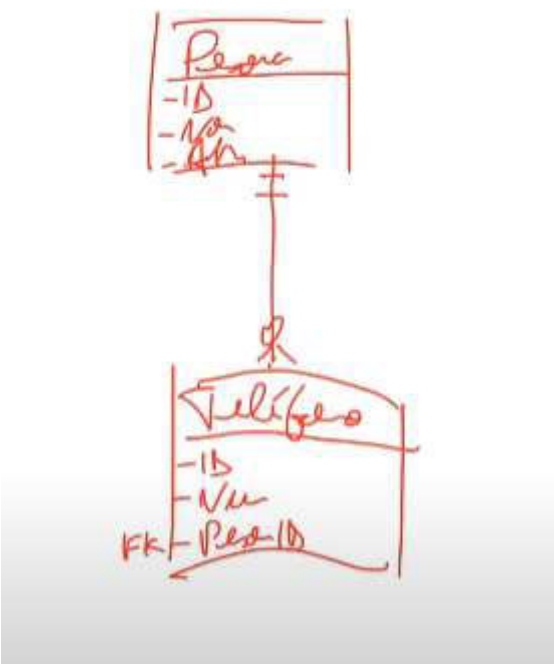
Agregación y composición:

Composición: vamos a tener clase Persona y otra clase teléfono con una relación de composición



Si se borra la persona, se borran los teléfonos.

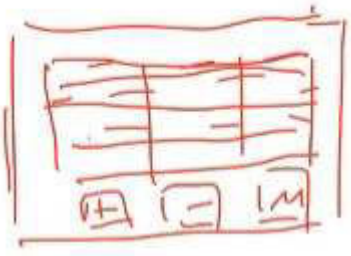
En la BD, vamos a tener dos tablas: persona y teléfono con una relación.



Persona puede tener 0 o muchos teléfonos y teléfonos va a tener una sola persona.

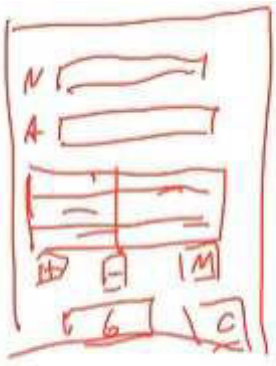
Composición y agregación tienen que estar manifiestas en el dominio.

Se debe poder tener una persona, guardarle un teléfono y después guardar todo.



Pantalla donde se vera a todas las personas con botón de alta, baja y modificar.

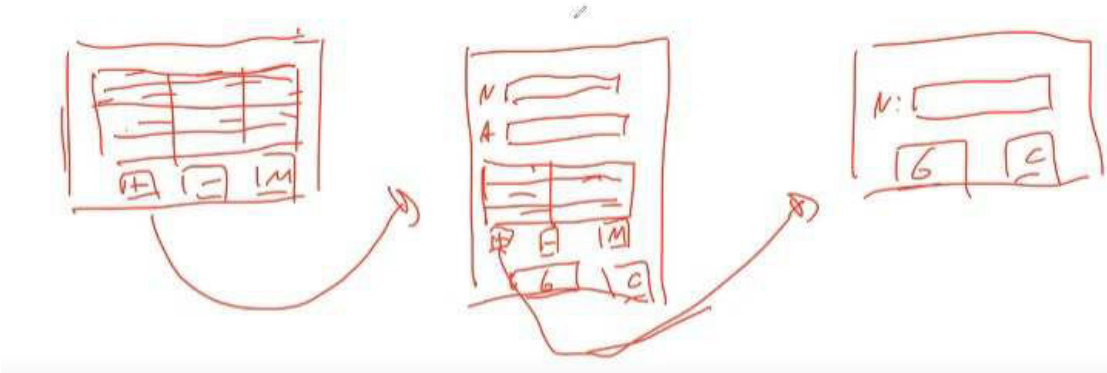
Toco algún botón y abre otra pantalla:



Nombre, apellido, teléfono. Y con los teléfonos también se puede agregar, quitar o modificar uno que este ahí.

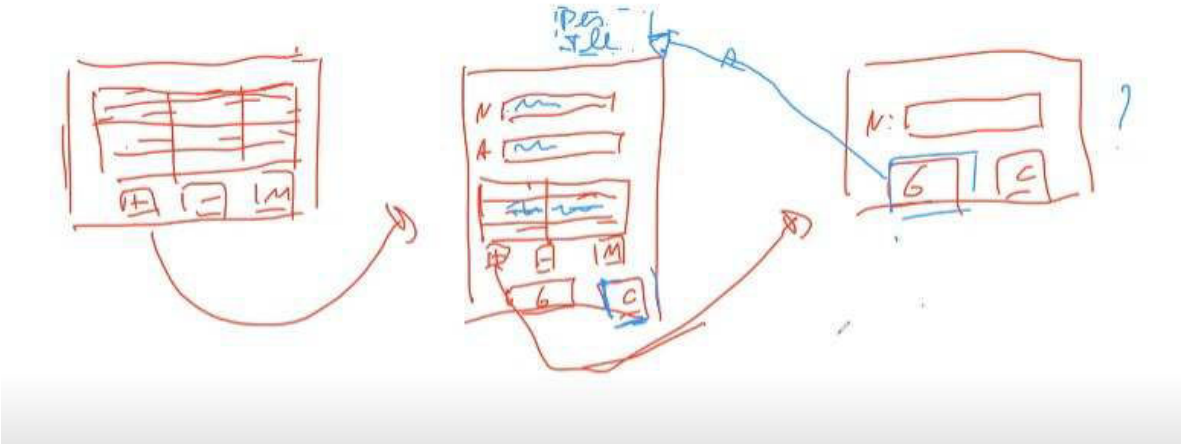
Otro botón guardar y cancelar.

Al modificar o agregar algún teléfono, se debe abrir una tercera ventana para editar los teléfonos con un botón de guardar y otro de cancelar.



¿Qué pasa cuando presiono el guardar ultimo? ¿El de teléfonos?

Ahí no se debe ir a la base. Ya sea nuevo o modificar, al tocar guardar al final, se debe agregar ese teléfono a la segunda pestaña.



El botón guarda el botón, a la persona. No a la BD. Y se actualiza la grilla de la ventana 2.

Ahora bien, **¿qué pasa si toco el de guardar en la 2da pestaña?**

Se guarda primero la persona para obtener el ID y luego se guarda el teléfono. (ALTA)

Primero persistir persona luego teléfonos.

¿Como se persiste los teléfonos?

Porque la persona puede tener 3 teléfonos y yo modifico uno solo. Entonces teléfono debería tener un estado, y al leer un estado me fijo si se creó o modifico.

Estado: sin cambios, agregado, modificado, eliminado, quitado

La diferencia entre quitado y eliminado es que, si edito una persona, agrego un teléfono y después lo saco es quitado.

Si selecciono un teléfono de la lista, lo modifico y le cambio el número, **¿cómo puedo saber que teléfono estoy editando?** De los que están en la colección que todavía no se persistió.

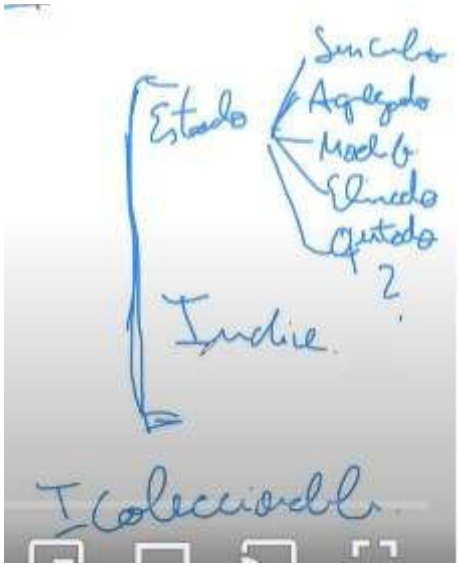
Para esto va a ser a través de un índice, los teléfonos van a tener además un índice entonces lo buscamos por su índice en la colección.

Si tengo una clase teléfono, pero para participar de agregación con persona, hay que agregarle un estado e índice al teléfono. Pero si persona está en agregación con institución, entonces le pongo estado persona y identificación. Después institución puede ser una clase en agregación con otra, ej. Asociación.

Si hay varias agregaciones hay que implementar Estado e índice.

Para cerrar entonces, se puede tener una interfaz: IColeccionable que obligue a implementar:

Estado e índice.



Entonces teléfono es icoleccionable, persona si es agregación con algo también es icoleccionable.

EJEMPLO PRACTICO:

Al guardar persona, se guarda primero a la persona

Y luego hay que guardar al teléfono.

```
public int Guardar(Persona pPersona)
{
    int mRet = PersonaDAL.Guardar(pPersona);
    GuardarTelefonos(pPersona);

    return mRet;
}
```

¿Como hago para acceder a los teléfonos de la persona en guardar teléfonos?

El for se hizo al revés porque si es de abajo para arriba, al encontrar uno y borrarlo si borro uno de la posición 2 el de la 3 pasa a la 2, y a esa la paso de largo.

Entonces hago al revés.

El eliminar, elimino primero al teléfono y luego a la persona.

Ahora vemos otra variante: **al problema de acceder a la colección privada de teléfonos de la persona.**

Desde persona tengo un atributo internal, TelefonosColeccion (internal es que podía ser visto desde el mismo proyecto) y yo necesito verlo desde otro. Persona está en BEL y necesito verlo en la BLL

Pero puedo agregar arriba:

```
//Para que el atributo "internal ObtenerelefonosColeccion"
//que nos permite acceder a la colección encapsulada,
//pueda ser vista desde la BL (para poder persistir los teléfonos) debemos agregar esta instrucción
[assembly: InternalsVisibleTo("BLL")]
```

Los internal también se verán dentro de la BLL.

```
internal List<Telefono> TelefonosColeccion
{
    get
    {
        return mTelefonos;
    }
}
```

Va a ser visible en ambos proyectos.

```
public void AgregarTelefono(Telefono pTelefono, Constantes.EstadosColeccion pEstado = Constantes.EstadosColeccion.Ag
{
    /*
    AL AGREGAR UN TELEFONO, NO LO AGREGAMOS EN LA BASE DE DATOS INMEDIATAMENTE
    SOLO LO AGREGAMOS EN LA COLECCION Y LE ASIGNAMOS EL ESTADO AGREGADO,
    DEBIDO A QUE PUEDE OCURRIR QUE EL USUARIO CANCELE LA OPERACION DE EDICION
    O ALTA DE LA PERSONA ANTES DE QUERER GUARDAR LOS CAMBIOS EN LA BASE

    INMEDIATAMENTE DESPUES DE AGREGADO HACEMOS QUE EL PROPIO TELEFONO CONOZCA SU POSICION
    DENTRO DE LA COLECCION (INDICECOLECCION) DEBIDO A QUE EL TELEFONO ES NUEVO Y NO POSEE ID
    TODAVIA POR NO EXISTIR EN LA BASE DE DATOS. EL INDICE SERA LA FORMA DE IDENTIFICARLO
    */

    mTelefonos.Add(pTelefono);
    pTelefono.EstadoColeccion = pEstado;
    pTelefono.IndiceColeccion =  mTelefonos.IndexOf(pTelefono);
}
```

Tiene un parámetro opcional: porque cuando listamos, ej. Cargar teléfonos

```
public Persona Obtener(int pId)
{
    Persona mPers = PersonaDAL.Obtener(pId);

    foreach(Telefono t in TelefonoDAL.Listar(pId))
    {
        mPers.AgregarTelefono(t, Constantes.EstadosColeccion.SinCambio);
    }
    return mPers;
}
```

Cuando obtenemos la persona y después le agregamos los teléfonos, no queremos que queden con estado agregado, entonces le agregamos el estado sin cambios.

Alumno forma parte de curso, curso forma parte de carrera, hay que reescribir toda la persistencia, **¿entonces para manejar todo esto que atributos del teléfono necesito?**

Estado e índice, y para la parte de persistir también, **estado índice**.

Se usa una colección genérica de tipo T, que será IColeccionable:

Se reusó toda la lógica de persistencia en una interfaz IColeccionable que a la vez es IPersistible.

Agregación:

Alumnos y cursos. Cualquiera puede tener una colección sobre la otra.

En estas colecciones uso leasy loading: si cargo todos los cursos y me traigo un alumno de cada curso, se hace una llamada recursiva y me traigo casi toda la BD a memoria.

Carga Perezosa: se carga un curso, pero al listar los cursos alumnos esta vacío, cuando en un curso le pido por los alumnos recién ahí pregunta si esta la colección cargada de alumnos y devuelvo.

La porx vuelve a preg si los tiene cargados y devuelve.

ABM alumnos, puedo agregar un nuevo alumno

Y después también puedo entrar a cursos y crear cursos, agregarle alumnos o editar.

Curso tiene lista de alumnos. Filtra por el estado a la colección.

Al guardar se guarda alumnos. Y al guardar alumnos y estos pueden estar solo agregados o eliminados.

Eliminados no hace delete de alumnos. Le pega a la tabla intermedia porque es una relación de muchos a muchos entonces necesito tabla intermedia.

Al guardar curso, guardo los alumnos.