

UNIDAD 1 - ENSAMBLADOS

Itinerario 2

Una de las características de las aplicaciones .NET es su distribución en forma de **ensamblados** que incluyen *ejecutables y componentes*.

Al compilar una aplicación .NET, se obtendrá un módulo **ensamblado**. Puede ser un EXE o una DLL según el tipo de proyecto que se esté desarrollando.

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

Su composición se agrupa en encabezados de archivos Windows PE, encabezado de archivo .NET framework, metadatos y código de lenguaje intermedio MSIL.

- ENCABEZADO

Todos los archivos EXE al igual que los módulos tienen un encabezado PE. Una de las referencias más importantes que **poseen es el código que se debe ejecutar cuando el SO los invoca**.

Los módulos contienen código MSIL, este es un **conjunto de instrucciones nativas comprendidas por la CPU destino**. Por ello, la primera instrucción de los módulos administrados es JMP (salto) que apunta al código MSIL.

- METADATOS

Generalmente se entiende un metadato como la **descripción de los datos**. Describen los **tipos a los cuales hace referencia el módulo actual**.

No existe posibilidad alguna de entregar un módulo sin sus metadatos o al revés. Puesto que los metadatos son indispensables en NET.

Desde NET se pueden leer los metadatos del módulo usando **Reflection**.

- LENGUAJE INTERMEDIO

MSIL presenta dos beneficios:

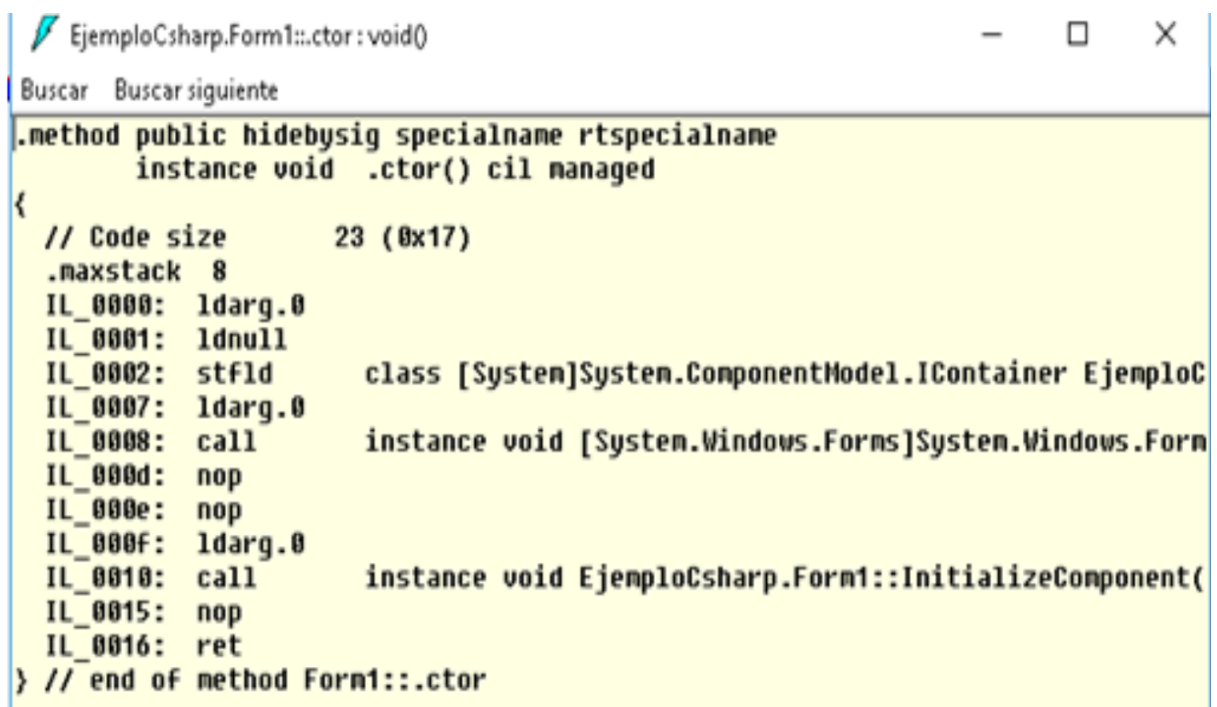
- 1) Proporciona un **lenguaje intermedio de fácil adaptación a las CPU del mercado**, razón por la cual se debe instalar el framework .NET en la PC donde se implemente el programa.
- 2) **Refuerza a nuestra aplicación de mayor seguridad en su ejecución** al hacer más robusto al código administrado.

Esto es un ejemplo de MSIL, consideremos el siguiente código de C# .NET en una app winforms.

```
private void Button1_Click(object sender, EventArgs e)
{
    int a = 0;
    int b = 0;
    int c = 0;

    a = Convert.ToInt32(TextBox1.Text);
    b = Convert.ToInt32(TextBox2.Text);
    c = a + b;
    TextBox3.Text = Convert.
```

Código MSIL resultante:



```
EjemploCsharp.Form1::ctor: void()
Buscar Buscar siguiente
.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    // Code size          23 (0x17)
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: ldnull
    IL_0002: stfld      class [System]System.ComponentModel.IContainer EjemploC
    IL_0007: ldarg.0
    IL_0008: call        instance void [System.Windows.Forms]System.Windows.Form
    IL_000d: nop
    IL_000e: nop
    IL_000f: ldarg.0
    IL_0010: call        instance void EjemploCsharp.Form1::InitializeComponent(
    IL_0015: nop
    IL_0016: ret
} // end of method Form1::ctor
```

ENSAMBLADOS

Un **ensamblado** es la mejor unidad de reuso en .NET. Es por ello que debe mantener juntos a los módulos que se complementen entre sí.

Otra característica es que posee **control de versiones** – aspecto muy importante para un correcto control de las aplicaciones. Los tipos contenidos en un ensamblado tienen los mismos permisos, si desea tener una configuración diferente deberá considerarlo al agrupar los módulos.

Hay dos tipos de ensamblados, **estáticos** y **dinámicos**. La mayoría de los compiladores de C# o vb.net usan ensamblados estáticos, estos se corresponden con uno o varios archivos físicos. Por su parte, los dinámicos, se crean en la RAM al momento de ser utilizados.

Dentro del ensamblado se encuentra un archivo importante, el **manifiesto**.

Este contiene los metadatos sobre el ensamblado. Datos como su versión, referencia cultural y la clave pública de la compañía que lo desarrolla y la lista de sistemas operativos y CPU donde se podrá ejecutar el ensamblado.

Posee también una colección de tablas que muestra la lista de archivos que forman un ensamblado.

Por defecto el runtime .NET busca los ensamblados que se encuentran en el mismo espacio de directorio de la aplicación que efectúa la llamada evitando con esto conflictos con los nombres de los archivos. Pero puede ocurrir, sobre todo cuando los ensamblados se comparten con otras aplicaciones, que los nombres entren en conflicto.

Para asegurar el nombre del ensamblado, puede generar una clave pública y privada aleatoria con la aplicación **SN**, con el siguiente comando.

```
Sn -k miclave.snk
```

Luego podrá firmar el ensamblado de la aplicación y así asegurar un nombre único para el ensamblado.

Visual basic: `Vbc modulo.vb /out:ensamblado.dll /keyfile:miclave.snk`

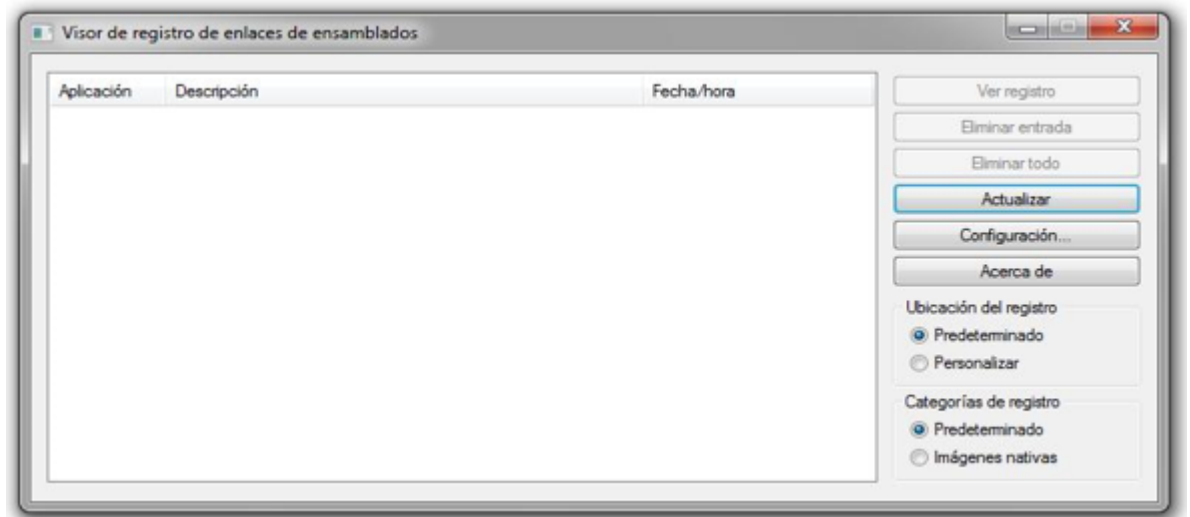
C#: `csc modulo.cs /out:ensamblado.dll /keyfile:miclave.snk`
Vbc `modulo.vb /out:ensamblado.dll /keyfile:miclave.snk`

HERRAMIENTAS

El framework .NET ofrece un conjunto de herramientas fuera del IDE de desarrollo que facilitan la lectura y la administración de los ensamblados.

1) **FUSLOGVW**

Es un **visor del registro de enlaces de los ensamblados**. Facilita la lectura de la localización de los ensamblados. Es una herramienta muy importante cuando es **necesario seguir errores ocurridos en los ensamblados**.



2) **GACUTIL**

Le **permite administrar el caché de ensamblados global (GAC)** desde la **línea de comandos**.

El caché global reemplaza el registro de las librerías en el modelo COM.

Al registrarlo en la GAC una aplicación podrá usar un ensamblado que no se encuentra en su carpeta.

Para instalar un ensamblado en el GAC solo debe ingresar:

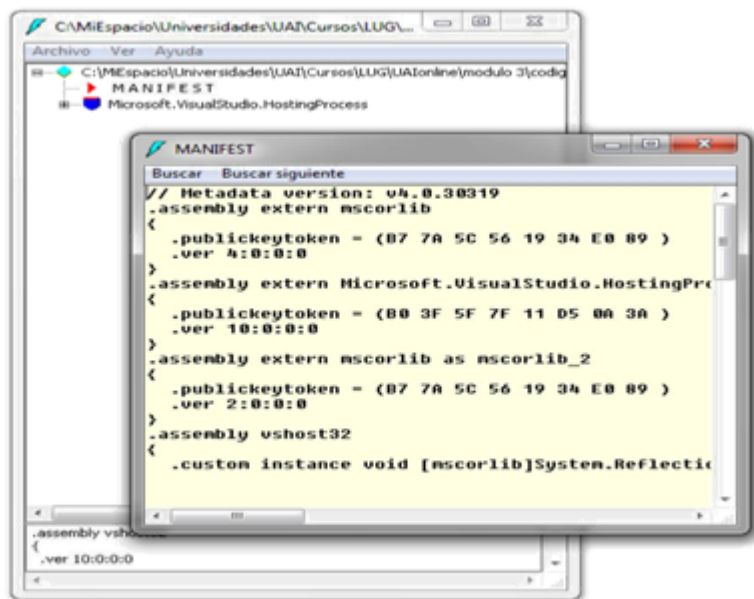
```
Gacutil /i modulo.dll
```

Para borrarlo de la GAC:

```
Gacutil -u modulo
```

3) **ILDASM**

Es una herramienta que nos permite ver el código administrado:

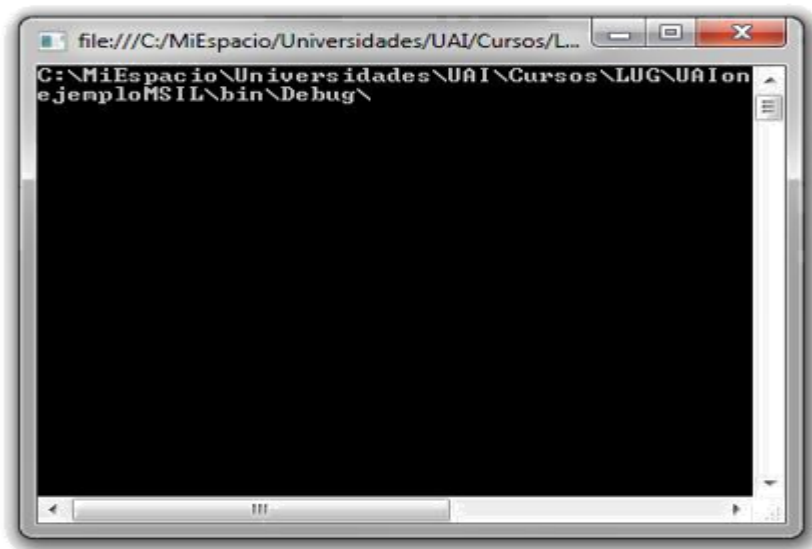


APPDOMAIN

Todas las aplicaciones de .NET corren en un **APPDOMAIN**.

La clase **APPDOMAIN** permite conocer detalles e información acerca del dominio donde se encuentra la aplicación.

Por ejemplo, podemos conocer el directorio donde se encuentra la aplicación.



El **código fuente** de este ejemplo es:

```
public void Main()
{
    AppDomain D = AppDomain.CurrentDomain;
    Console.WriteLine(D.BaseDirectory);
    Console.ReadKey();
}
```

Se crea una variable de nombre **D** que asume el rol del dominio donde se ejecuta la aplicación. Luego usa la propiedad **BaseDirectory** para obtener información.

Otras propiedades que puede utilizar son:

FriendlyName: Nombre descriptivo del dominio

ShadowCopyFile: indica si todos los ensamblados fueron copiados en primer lugar en otro directorio.

UNIDAD 2 - ARQUITECTURA

Itinerario 3

CONCEPTO DE ARQUITECTURA

Se deben tener en cuenta varios conceptos cuando se habla de arquitectura:

- **Vista estructural de alto nivel**: La arquitectura de un sistema describe en un alto nivel de abstracción como se encuentra estructurado el mismo, que servicios expone tanto interna como externamente y cómo se comunican dichos servicios entre sí.
- **Se concentra en requerimientos no funcionales**: Los requerimientos NF definen la arquitectura en gran parte.
- **Balance**: Una estructura tiene que estar balanceada.
- **Es esencial para el éxito o fracaso de un proyecto**: Una arquitectura errónea podría provocar como producto un sistema difícil de mantener, escalar y que no logre los requerimientos de disponibilidad y seguridad necesarios.

¿CÓMO DEFINIR UNA ARQUITECTURA DE SOFTWARE?

Se debe tener en cuenta:

- 1) **Identificar los requerimientos no funcionales**: Ejemplo: “Disponibilidad” si tengo que asegurar un alto grado de disponibilidad de una aplicación es muy probable que tenga que distribuir entre varios servidores para que esté disponible y no tenga problemas de rendimiento. Arquitectura distribuida.
- 2) **Identificar interacciones con otros sistemas**: Cómo se va relacionar con otros sistemas, y a través de que tecnologías.
- 3) **Planificar la evolución del sistema**: Identificar las partes mutables e inmutables del sistema, así como los costos de los posibles cambios.
- 4) **Definir las tecnologías a implementar**

TIPOS DE ARQUITECTURA DE SOFTWARE

1) **CLIENTE SERVIDOR**

En una arquitectura Cliente/Servidor **los datos son compartidos en un servidor central.**

Se pueden compartir archivos e impresoras, pero la entrada, procesamiento y salida de la información se continúa realizando en la misma PC del cliente.

2) **CAPAS / LAYERS**

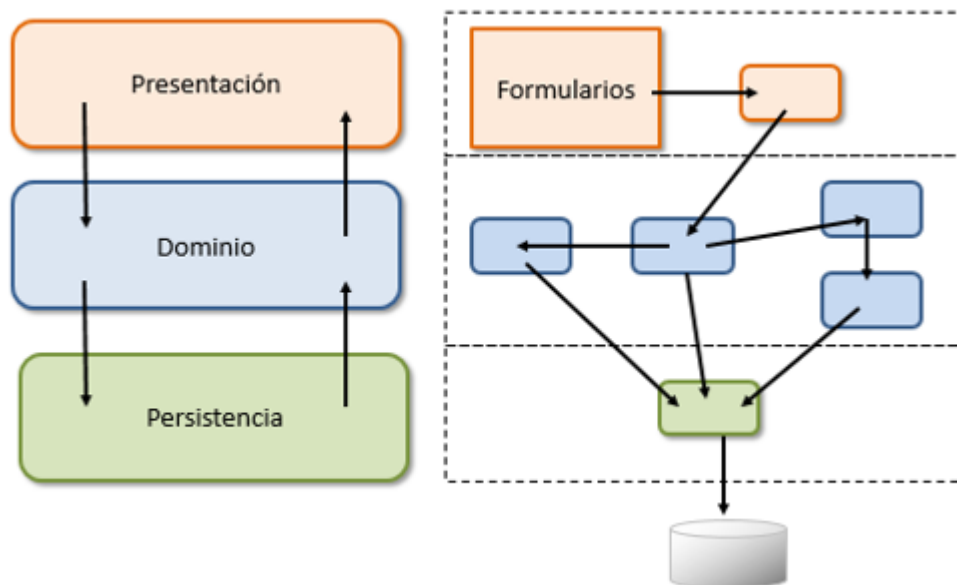
Las capas lógicas son agrupaciones de componentes que exponen una funcionalidad de alto nivel (por ejemplo: la BLL puede estar conformada por el subsistema de control de inventario).

Las capas lógicas **se comunican entre sí por medio de interfaces** y cada interfaz de cada capa expone los servicios de la misma.

Las Capas (Layers) **se ocupan de la división lógica de componentes y funcionalidad**, y no tiene en cuenta la localización física de componentes en diferentes servidores.

Por el contrario, los Niveles (Tiers) se ocupan de la distribución física de componentes y funcionalidad en servidores separados, teniendo en cuenta topología de redes y localizaciones remotas.

A) **ARQUITECTURA EN 3 CAPAS**



Presentación: Encargada de cómo se deben mostrar los datos en las GUI. Además, es la responsable de capturar las entradas del usuario e invocar a los servicios necesarios para llevar a cabo las funciones requeridas.

Dominio/Negocio: Contiene las reglas del negocio.

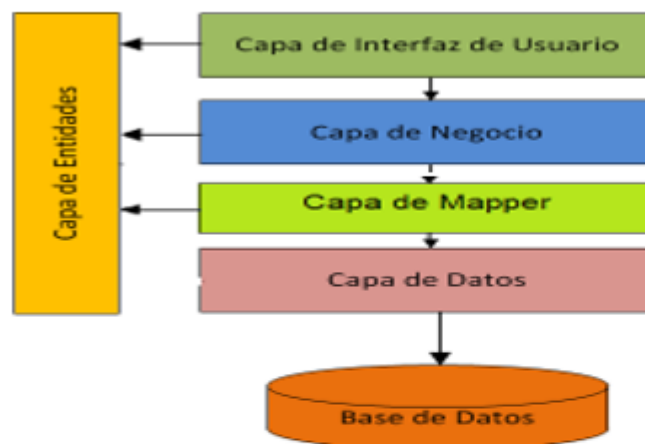
Persistencia: Es la capa que se encarga de que los objetos persistan entre ejecuciones del sistema. El mecanismo de persistencia puede ser una base de datos relacional, archivos XML, etc.

B) ARQUITECTURA EN n CAPAS

Es la segmentación de una aplicación en partes y la distribución de esas partes a lo largo de una red.

La ventaja de utilizar este tipo de arquitectura está en que la capacidad de ejecutar, mantener y mejorar aplicaciones de gran escala aumenta considerablemente.

Un ejemplo de N-capas, siendo N=5.



Interfaz de Usuario (UI -User Interfaces): Es la capa de presentación, con la que interactúan los usuarios.

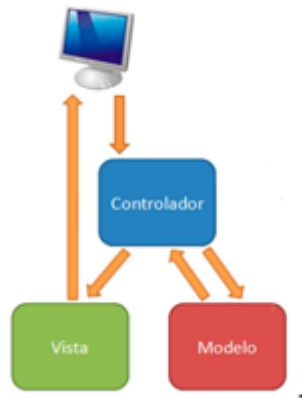
Entidades de Negocio o BE (Business Entity): se utiliza para la comunicación o medio de transporte entre las demás capas para llevar y traer datos, regularmente a través de parámetros (solamente las propiedades de las clases)

Negocio (BLL -Business Logic Layer): Esta capa contiene toda la lógica del negocio.

Mapper (MPP): Capa que convierte objetos a datos relacionales y viceversa. Posee la lógica para la transformación.

Datos (DAL -Data Access Layer): Es la capa encargada de la persistencia en la base de datos

C) **ARQUITECTURA MVC (Modelo Vista Controlador)**



Es un estilo de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.

El **MODELO** contiene una representación de los datos que maneja el sistema, su lógica de negocio y sus mecanismos de persistencia.

La **VISTA** ó Interfaz de usuario, compone la información que se envía al cliente y los mecanismo de interacción con éste.

El **CONTROLADOR** actúa como intermediario entre el Modelo y la Vista, gestionando el flujo de información entre ellos y las transformaciones para adaptar los datos a las necesidades de cada uno.

UNIDAD 3 - ACCESO A DATOS - MODO DESCONECTADO Y CONECTADO

Itinerario 4

BASE DE DATOS RELACIONAL

Las BD relacionales son los motores que cumplen con el modelo relacional. Este modelo se conforma por entidades y sus interrelaciones.

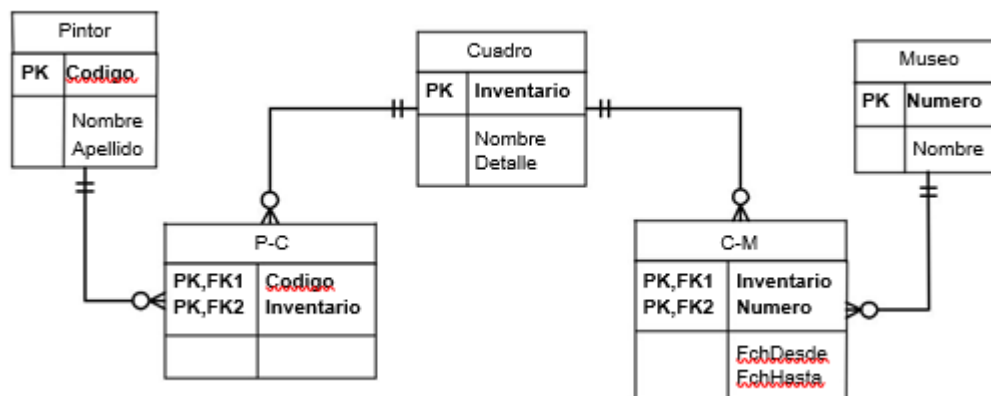
Las **ENTIDADES** son el conjunto de campos que conforman una tabla de la BD, generalmente expresadas con un nombre singular (Ejemplo: "Cliente").

Los **CAMPOS** son los descriptores de dicha entidad, generalmente representados por un nombre y un tipo de dato (Ejemplo: Nombre de tipo texto).

El **OBJETIVO** primario de las bases de datos es evitar la duplicación innecesaria de la información razón por la cual no toda la información puede estar en una sola entidad.

Las **interrelaciones** son las asociaciones que existen entre las entidades y expresan la vinculación de datos.

Ejemplo de BD:



Para acceder a los datos en una tabla se utiliza SQL (Standard Query Language) el cual está formado por tres grupos básicos:

- 1) La selección de los campos.
- 2) Las tablas que participan.
- 3) Las condiciones de filtrado.

Un ejemplo de una consulta podría ser: **Select * from Cliente**

NOTA: Para más consultas, leer el manual en PDF.

Itinerario 5

ENTORNO CONECTADO

Un entorno conectado es aquel en que los usuarios están conectados continuamente a una fuente de datos.

VENTAJAS	DESVENTAJAS
El entorno es más fácil de mantener.	Debe existir una conexión de red constante.
La concurrencia se controla fácilmente.	Escalabilidad limitada.
Datos más actualizados.	

ENTORNO DESCONECTADO

Un entorno desconectado es aquel en el que los datos pueden modificarse de forma independiente y los cambios se escriben posteriormente en la base de datos.

VENTAJAS	DESVENTAJAS
Las conexiones se utilizan durante el menor tiempo posible.	Los datos no siempre están actualizados.
Mejora la escalabilidad y el rendimiento.	Pueden producirse conflictos de cambio.

ADO.NET

ADO.NET es un conjunto de componentes del software que pueden ser usados por los programadores para acceder a datos y a servicios de datos. Es parte de la biblioteca de clases base que están incluidas en el Microsoft .NET Framework.

Proveedores de datos

Los proveedores de datos facilitan la comunicación con los motores de BD. Además, brindan los medios de comunicación necesarios para que dos aplicaciones puedan convivir en escenarios totalmente diferentes.

ADO.NET propone tres proveedores distintos:

1. **OLEDB**
2. **SQL Server**: Es un proveedor escrito exclusivamente para base de datos SQL. La comunicación entre el programa y la base de datos es directa.
3. **ODBC**

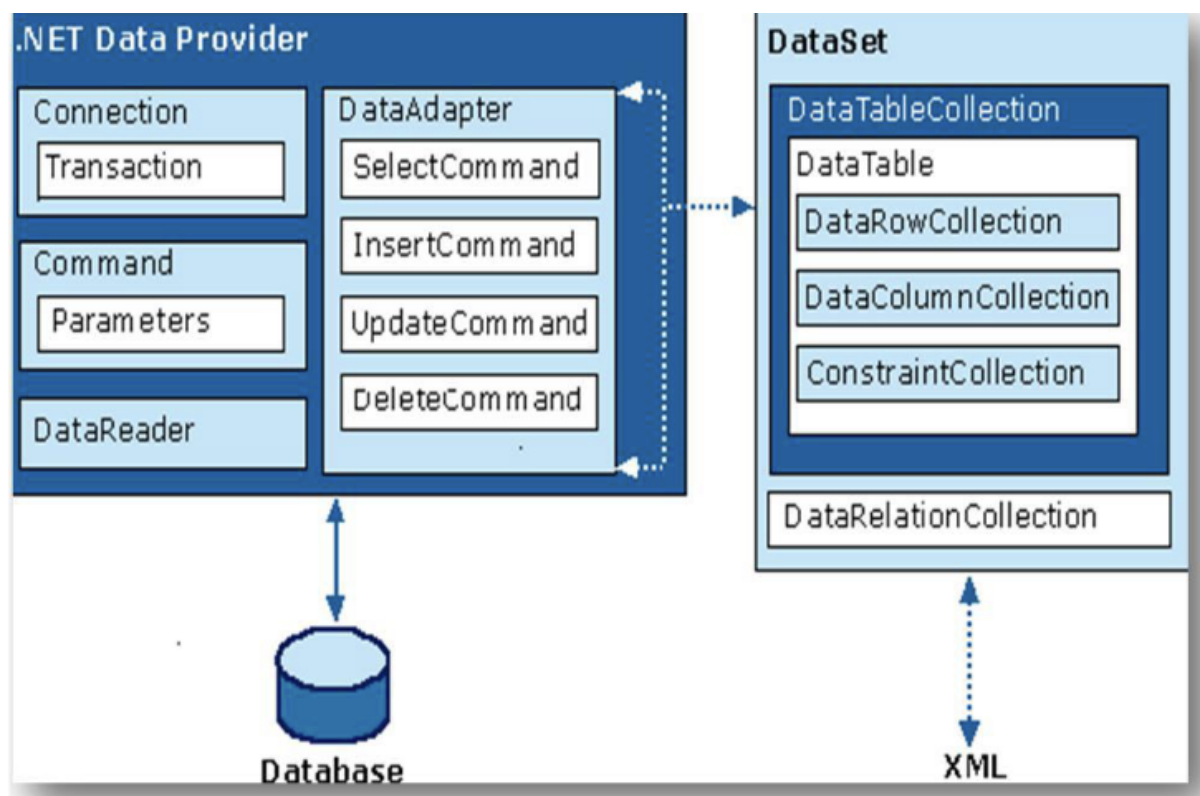
Arquitectura ADO.NET

ADO.NET representa la relación entre todos los objetos que permiten la conexión a las bases de datos.

Dentro de ADO.NET tenemos los objetos **Connection**, **Command**, **Adapter**, **dataReader**.

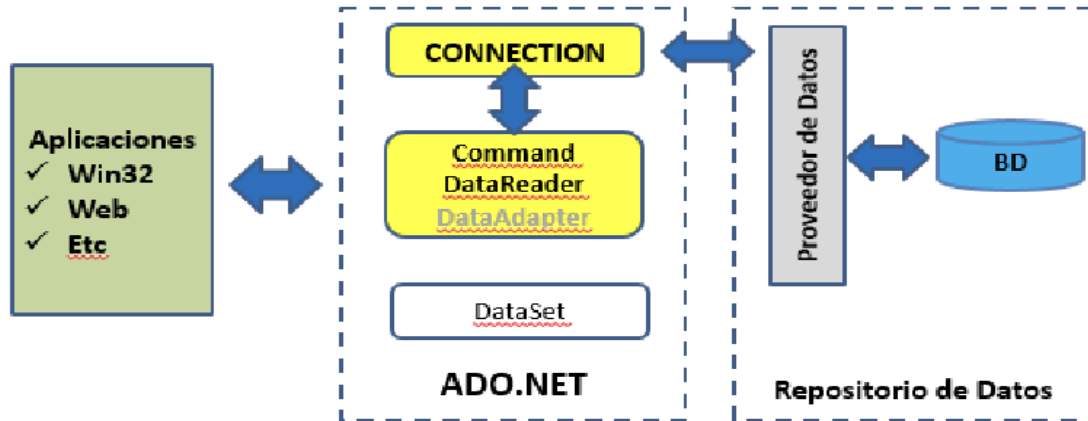
Si trabajamos con SQL Server, los renombramos a **SqlConnection**, **SqlCommand**, **sqlAdapter**, **sqlDataReader**.

Fuera del modelo tenemos el **DataSet**.



MODO CONECTADO

Un esquema del modelo conectado



1) OBJETO SQLCONNECTION

El objeto **SqlConnection** se encuentra dentro del espacio de nombre **System.data.sqlCliente** y permite la conexión a la base de datos.

Ejemplo de string de conexión:

```
Data Source=[servidor];Initial Catalog=[Nombre de BD];Integrated Security=True;
```

Este es el objeto que más se necesita si se utiliza el entorno conectado ya que determina la conexión.

Para trabajar en el modo conectado, utilizaremos una propiedad y dos métodos del objeto.

La propiedad es **ConnectionString** donde se le carga el string previamente mencionado.

El método **Open** es la acción que dispara el objeto para conectarse a la BD.

El método **Close** es la acción que cierra la conexión a la BD para liberar recursos.

```
Using System.Data;
Using System.Data.SqlClient;

SqlConnection oCn = new SqlConnection();
oCn.ConnectionString = ".....";
oCn.Open();

(.....)
oCn.Close();
```

2) OBJETO SQLCOMMAND

Este objeto se complementa con el objeto de conexión y **tiene la responsabilidad de ejecutar todas las órdenes necesarias para tomar los datos de la base**. (Operaciones de **Agregar**, **Modificar**, **Borrar** y **Seleccionar**).

Generalmente **se usa con sintaxis SQL pero también puede ejecutar procedimientos almacenados** (Funciones precompiladas en la BD). Se utilizan con más frecuencia ya que mitigan los ataques por SQL Injection.

Para ejecutar un **command** se debe respetar los siguientes pasos:

1. Relacionar el comando con la conexión
2. Especificar el tipo de comando (texto, procedimiento almacenado)
3. Facilitar el comando a ejecutar

```
SqlCommand oCmd = new SqlCommand();  
oCmd.Connection = oCn;  
oCmd.CommandType = CommandType.Text;  
oCmd.CommandText = "select * from Pintor";
```

Luego, debemos ejecutar el comando. La ejecución puede ser de tres formas diferentes:

1. Ejecutar el comando y devolver un conjunto de datos.
2. Ejecutar el comando y devolver un escalar.
3. Ejecutar el comando y no devolver nada.

```
1. oCmd.ExecuteReader();  
2. oCmd.ExecuteScalar();  
3. oCmd.ExecuteNonQuery();
```

3) OBJETO SQLDATAREADER

El objeto **DataReader** **representa una tabla**. Si bien es muy rápido, tiene algunas limitaciones como por ejemplo que puede recorrer en una única dirección.

Posee un método llamado **Read** que lee el registro y pasa al puntero siguiente.

```
SqlDataReader oDr= cmd.ExecuteReader();
while (oDr.Read())
{
    ....
}
oDr.Close();
```

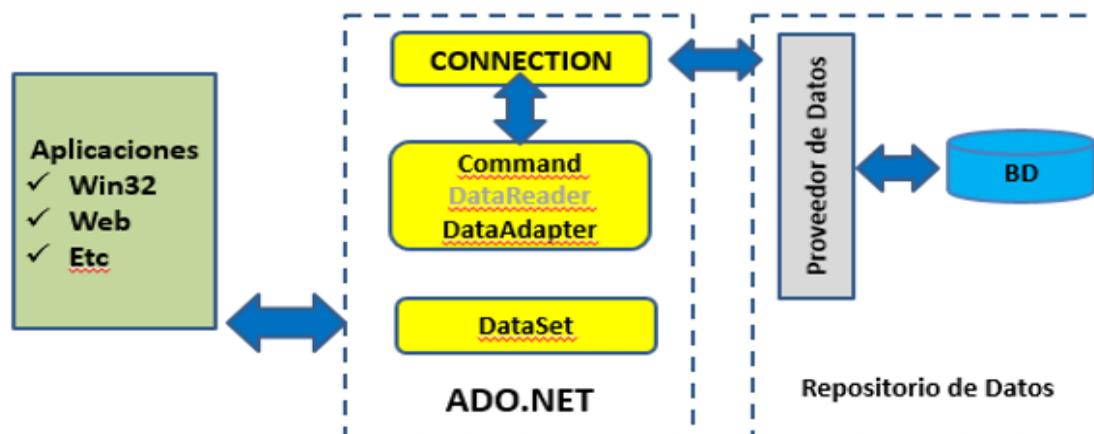
El **dataReader** necesita del **SqlCommand**, es decir, al ejecutar el método **ExecuteReader** del objeto **SqlCommand** obtenemos un objeto **dataReader**.

```
SqlConnection oCn = new System.Data.SqlClient.SqlConnection();
oCn.ConnectionString = "...";
oCn.Open();

SqlCommand oCmd = new System.Data.SqlClient.SqlCommand();
oCmd.Connection = oCn;
oCmd.CommandType = CommandType.Text;
oCmd.CommandText = "select * from pintor";
SqlDataReader oDr = oCmd.ExecuteReader();
If (oCn.State == ConnectionState.Connecting)
{ oCn.Close(); }
```

MODO DESCONECTADO

Uno de los problemas principales que tiene es la **sincronización** de los datos, puesto que si alguien cambió los datos mientras se estaba usando no sabremos cual es el valor definitivo.



1) OBJETO DATASET

El objeto **DataSet** no forma parte del modelo ADO.NET pero se complementan ya que puede funcionar como una base de datos.

Contiene **DataTables** que son tablas vinculadas al modelo de datos y **DataRelation** que son las relaciones entre las tablas.

A su vez, el **datatable** está formado por **datarows** y estas a su vez por **datacolumns**.

La **ventaja** del **DataSet** es que está formado por colecciones enlazadas que se pueden recorrer fácilmente con un **foreach**.

```
DataSet Ds = new DataSet();
DataTable Dt = new DataTable();
Dt.TableName = "Persona";
//Creo la columna Id
    DataColumn ColId ;

//Completo la columna Id
    ColId = new DataColumn("ID", typeof(int));

//Creo la columna Nombre
    DataColumn ColNombre = new As DataColumn();
//Completo la columna ColNombre
ColNombre.ColumnName = "Nombre";
ColNombre.DataType = typeof(string);
ColNombre.MaxLength = 50;

//Agrego las columnas a la tabla
    Dt.Columns.Add(ColId);
```

```

Dt.Columns.Add(ColNombre);

//Creo una fila con los datos en las columnas
DataRow Dr;
Dr = Dt.NewRow();
Dr["Id"] = 1;
Dr["Nombre"] = "Juan";
Dt.Rows.Add(Dr);

Dr = Dt.NewRow();
Dr["ID"] = 2;
Dr["Nombre"] = "Pedro";
Dt.Rows.Add(Dr);

//Determino la clave primaria

Dt.PrimaryKey = new DataColumn[] { Id };
Ds.Tables.Add(Dt);

this.cmbDataset.DataSource = Ds.Tables[0];
this.cmbDataset.DisplayMember = "Nombre" ;
this.cmbDataset.ValueMember = "Id";

```

Los pasos para crear un dataset sin una base de datos son:

- 1- Crear un objeto **Dataset**
- 2- Crear un **Datatable**
- 3- Crear el o los objetos **datacolumn** que se necesiten determinando el nombre del campo y el tipo de datos.
- 4- Agregar al **Datatable**
- 5- Llenar la colección de claves de la tabla con los campos que la componen
- 6- Agregar el objeto **Datatable** al **DataSet**
- 7- De existir, crear todas las relaciones.

2) OBJETO ADAPTER

Este objeto es el nexo entre el modelo ADO.NET con el objeto **DataSet**. Hay un tipo **DataAdapter** para cada proveedor de datos.

El DataAdapter tiene una sobrecarga en su constructor, entre sus opciones puede no tener argumentos o agregar argumentos.

Ejemplo de una rutina donde utilizamos el **SqlDataAdapter** para llenar un **dataSet**.

```
DataSet Ds = new DataSet();
SqlCommand oCmd = new SqlCommand();
SqlDataAdapter oDa = new SqlDataAdapter();
oCmd.Transaction = TX;
oCmd.Connection = oCnn;
oCmd.CommandType = CommandType.Text;
oCmd.CommandText = Query;
oDa.SelectCommand = oCmd;
oDa.Fill(Ds);
oDa.Dispose();
oCmd.Dispose();
Return Ds;
```

Itinerario 6

TRANSACCIONES

Las **transacciones** es una forma de controlar si la actualización de datos en una BD se deben aceptar o descartar.

¿Cómo usar las transacciones?

Las transacciones tienen sentido si vamos a realizar varias tareas de actualización (actualización, inserción, borrado).

Por ejemplo si intervienen varias tablas, ya que podemos actualizar los datos de una tabla y al intentar actualizar otra, es cuando se produce un error.

Podemos deshacer los cambios realizados en la primera tabla y cancelar toda la operación, de esta forma nos aseguramos que no queden datos 'colgados' y que solo tienen sentido si se realiza el proceso completo.

Propiedades de las Transacciones

Se las conoce como **ACID**:

1. **ATOMICIDAD**: Una transacción debe ser una unidad atómica de trabajo, o se hace todo o no se hace nada.
2. **COHERENCIA**: Debe dejar los datos en un estado coherente luego de realizada la transacción.
3. **AISLAMIENTO**: Las modificaciones realizadas por transacciones son tratadas en forma independiente, como si fueran un solo y único usuario de la BD.
4. **DURABILIDAD**: Una vez concluida la transacción, sus efectos son permanentes y no hay formas de deshacerlos.

¿Cómo hacer una transacción?

1. Abrimos la conexión.
2. Creamos el objeto **Transaction** mediante el método **BeginTransaction** de esa conexión.
3. Creamos los comandos.
4. Le asignamos a la propiedad **Transaction** de cada comando ese objeto que te habrá devuelto el método **BeginTransaction** (y que habrás guardado en una variable del tipo **Transaction**).
5. Usamos esos comandos.
6. Si todo ha ido bien, llamamos al método **Commit** del objeto **Transaction**.
7. En caso de que haya algún error o quieras cancelar, simplemente llamas al método **Rollback** de ese mismo objeto **Transaction**.

Isolation Level

Indica el comportamiento de bloqueo de la transacción.

Chaos	Los cambios pendientes de las transacciones más aisladas no se pueden sobrescribir.
ReadCommitted	Los bloqueos compartidos se mantienen mientras se están leyendo los datos para evitar lecturas erróneas. Sin embargo, es posible cambiar los datos antes del fin de la transacción, lo que provoca lecturas no repetibles o datos fantasma.
ReadUncommitted	Se pueden producir lecturas erróneas, lo que implica que no se emitan bloqueos compartidos y que no se cumplan los bloqueos exclusivos.
RepeatableRead	Los bloqueos se realizan en todos los datos utilizados en una consulta para evitar que otros usuarios los actualicen. Esto evita las lecturas no repetibles pero sigue existiendo la posibilidad de que se produzcan filas fantasma.

<u>Serializable</u>	Se realiza un bloqueo de intervalo en <u>DataSet</u> , lo que impide que otros usuarios actualicen o inserten filas en el conjunto de datos hasta que la transacción haya terminado.
<u>Snapshot</u>	Reduce el bloqueo almacenando una versión de los datos que una aplicación puede leer mientras otra los está modificando. Indica que de una transacción no se pueden ver los cambios realizados en otras transacciones, aunque se vuelva a realizar una consulta.
<u>Unspecified</u>	Se utiliza un nivel de aislamiento distinto al especificado, pero no se puede determinar el nivel. Al utilizar <u>OdbcTransaction</u> , si no establece <u>IsolationLevel</u> o establece <u>IsolationLevel</u> en <u>Unspecified</u> , la transacción se ejecuta según el nivel de aislamiento predeterminado del controlador ODBC subyacente.

9

PROCEDIMIENTOS ALMACENADOS

Un **procedimiento almacenado** (stored procedure) es un procedimiento el cual es almacenado físicamente en una base de datos.

¿Por qué utilizar procedimientos almacenados?

- Programación modular.
- Distribución del trabajo.
- Seguridad de la BD.
- Ejecución más rápida.
- Reduce el tráfico de red.
- Proporciona flexibilidad.

VENTAJAS	DESVENTAJAS
Rendimiento: Al ser ejecutado por el motor de la BD, no es necesario transportar datos a ninguna parte.	Esclavitud: Una BD con muchos procedimientos almacenados es imposible de migrar a otro motor.
Potencia: Permiten ejecutar operaciones complejas en pocos pasos.	
Centralización: Están en un lugar centralizado y pueden ser ejecutados por cualquier aplicación que tenga acceso a la misma.	

CRIPTOGRAFÍA

La **criptografía** es un mecanismo que codifica un mensaje de manera tal que solo el emisor y el receptor autorizado pueden comprenderlo.

Utilidad

- **Confidencialidad**: Ayudan a proteger la identidad de un usuario.
- **Integridad de los datos**: Para ayudar a evitar su alteración.
- **Autenticación**: Para garantizar que los datos provienen de una parte concreta.
- **Sin rechazo**: Para evitar que una determinada parte niegue que envió un mensaje.

Tipos de Criptografía

Método	Descripción
Cifrado de clave secreta (criptografía simétrica)	Realiza la transformación de los datos para impedir que terceros los lean. Este tipo de cifrado utiliza una clave secreta compartida para cifrar y descifrar los datos.
Cifrado de clave pública (criptografía asimétrica)	Realiza la transformación de los datos para impedir que terceros los lean. Este tipo de cifrado utiliza un par de claves pública y privada para cifrar y descifrar los datos.
Firmas criptográficas	Ayuda a comprobar que los datos se originan en una parte específica mediante la creación de una firma digital única para esa parte. En este proceso también se usan funciones hash.

Valores hash criptográficos Asigna datos de cualquier longitud a una secuencia de bytes de longitud fija. Los valores hash son únicos estadísticamente; el valor hash de una secuencia de dos bytes distinta no será el mismo.

Criptografía Simétrica y Asimétrica



Longitudes de claves simétricas	Longitudes de claves asimétricas
64 bits	512 bits
80 bits	768 bits
112 bits	1792 bits
128 bits	2304 bits

VS Criptografía

Cifrado de clave secreta
(criptografía simétrica)

AesManaged (introducida en .NET Framework 3,5).
DESCryptoServiceProvider.
RC2CryptoServiceProvider.
RijndaelManaged.
TripleDESCryptoServiceProvider.

Cifrado de clave pública
(criptografía asimétrica)

DSACryptoServiceProvider
RSACryptoServiceProvider
ECDiffieHellman (clase base)
ECDiffieHellmanCng
ECDiffieHellmanCngPublicKey (clase base)
ECDiffieHellmanKeyDerivationFunction (clase base)
ECDsaCng

Firmas criptográficas

DSACryptoServiceProvider
RSACryptoServiceProvider
ECDsa (clase base)
ECDsaCng

Valores hash criptográficos

HMACSHA1.
MACTripleDES.
MD5CryptoServiceProvider.
RIPEMD160.
SHA1Managed.
SHA256Managed.
SHA384Managed.
SHA512Managed.

Itinerario 10

XML

XML es un lenguaje descriptivo ya que se utiliza para describir datos. Además, facilita la lectura y la comprensión.

Archivo bien formado

Cuando un archivo XML cumple con las reglas estandarizadas por la W3C se dice que es un **archivo XML bien formado**.

Algunas reglas definen que:

Todo archivo XML comienza con la directiva de procesamiento

<?xml versión="1.0"?>

Todos los nodos se forman por tres partes: apertura + valor + cierre

<Nombre>Pepe</Nombre>

Si el nodo no tiene valor se representa como un nodo vacío

<Nombre />

Todos los **atributos** deben estar en la marca de apertura y entre comillas

<Persona Ndni="20555888">

Siempre debe tener un nodo Raíz

Todos los nodos menos el raíz deben estar dentro de otro nodo

Tomando lo considerado anteriormente estamos en condiciones de mostrar un ejemplo de XML

```
<?xml version="1.0" encoding="utf-8" ?>
<materia nombre="LUG">
  <Alumno Legajo="1">
    <Nombre>Martin</Nombre>
    <Apellido>Belgrano</Apellido>
  </Alumno>
  <Alumno Legajo="2">
    <Nombre>Angela</Nombre>
    <Apellido>Sarmiento</Apellido>
  </Alumno>
</materia>
```


DTD y Esquemas

Los DTD y Esquemas se utilizan para validar la estructura de los archivos XML, indicando por ejemplo si los nodos pueden repetirse, si son obligatorios, los nombres que deben usarse, etc.

1) DTD

Una característica es que su sintaxis es diferente al modelo usado por XML. Además, no permite manejar tipos de datos.

Los nodos se describen con una palabra reservada llamada ELEMENT (note que está en mayúsculas).

Los elementos están formados por un nombre y una regla, respetando la sintaxis ELEMENT nombre (regla). Las reglas pueden ser los nodos que contiene o el valor.

Existen tres tipos de reglas cuando no son nodos:

#PCDATA
ANY
EMPTY

Un ejemplo de un conjunto de definiciones de reglas en un archivo DTD es el que le mostramos a continuación:

```
<! ELEMENT pelicula (titulo, reparto?)>
```

```
<! ELEMENT titulo(#PCDATA)>
```

```
<! ELEMENT reparto(interprete, interprete)>
```

```
<! ELEMENT intérprete (#PCDATA)>
```

2) ESQUEMA

Son archivos con extensión con XSD que también define reglas para validar un archivo XML.

Permiten manejar tipos de datos como *string*, enteros, entre otros habilitando un mayor nivel de detalle en la descripción de las reglas.

Los elementos se describen con la palabra reservada **elemento**, ya habrá notado que en este caso es todo en minúsculas. Recuerde que cuando usamos DTD se escribe en mayúsculas.

La sintaxis para describir un **nodo** es similar a la siguiente:

<element name="pelicula" type="string">

Permite armar grupos que describen las relaciones de jerarquías entre nodos.

```
<element name="pelicula">
  <complexType>
    <all>
      <element name="titulo" type="string"/>
      <element name="minutos" type="integer"/>
    </all>
  </complexType>
</element>
```

Los **atributos**, tienen una sintaxis similar usando la palabra reservada *attribute*.

Veamos algunas formalidades para destacar es el uso de tres palabras reservadas:

- 1) *All*= Todos los elementos
- 2) *Sequence*= Todos los elementos respetando el orden
- 3) *Choice*= Es necesario incluir uno de los elementos

```
<?xml version="1.0"?>

<schema xmlns=http://www.w3.org/2001/XMLSchema> <element name="pelicula">
  <complexType>

    <sequence>
      <element name="titulo" Type="string"
    </sequence>
    </complexType>

  </element>
</schema>
```

CONCLUSIONES

DTD	XML Schema
Basadas en una sintaxis especializada.	Basados en XML
Compactas	No compactas
Hay muchas herramientas disponibles para procesar documentos XML con las DTD	Hay muy pocas herramientas para los esquemas XML
Tratan todos los datos como cadenas o cadenas enumeradas	Soportan una serie de tipos de datos (int, flota, boolean, date, etc.)
Emplean un modelo de datos cerrados que no permite prácticamente la ampliación.	Presentan un modelo de datos abierto, lo cual permite ampliar los vocabularios y establecer relaciones de herencia entre los elementos sin invalidar a los documentos.
Sólo permiten una asociación entre un documento y su DTD a través de la declaración de tipos de documentos.	Soportan la integración de los espacios de nombres, lo que permite asociar nodos individuales en un documento con las declaraciones de tipos de un esquema.
Soportan una forma primitiva de agrupación a través de entidades de parámetros, lo que supone una gran limitación, ya que el procesador XML no sabe nada acerca del agrupamiento.	Soportan los grupos de atributos, que le permiten combinar atributos lógicamente.

UNIDAD 4 - CONTROLES Y REPORTES

Itinerario 12

CONTROLES DE USUARIO

Cuando desarrollamos aplicaciones de escritorio, podemos crear controles personalizados para mejorar la calidad de nuestros desarrollos.

La **ventaja** de personalizar los controles es que se pueden **extender las funcionalidades de los controles nativos de la plataforma de programación**.

La clase **Control** es la clase base de los controles de formularios **WinForms**. Ofrece la interface necesaria de los controles básicos de .NET y toda la lógica para la presentación visual del control.

Las tareas son las siguientes según las fuentes de MSDN:

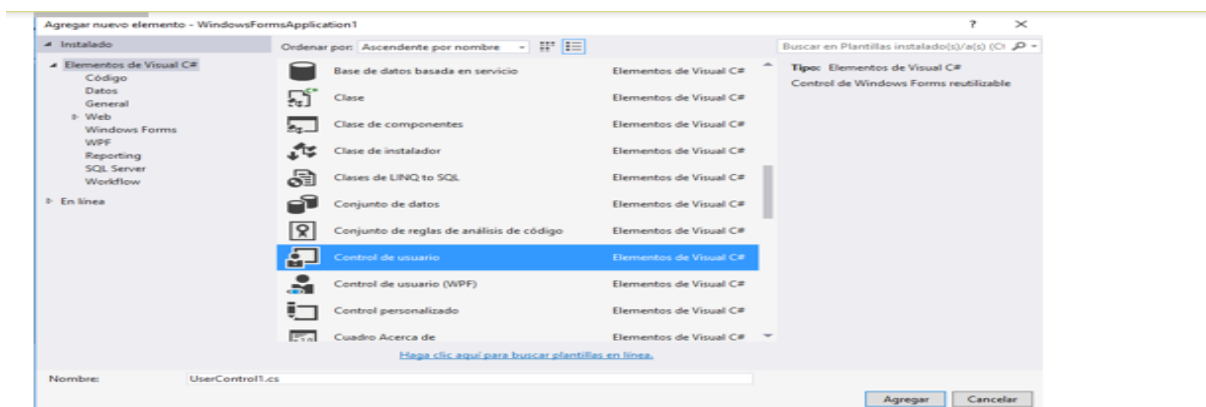
1. Expone un controlador de ventanas.
2. Administra el enrutamiento de mensajes.
3. Proporciona eventos del mouse (ratón) y del teclado y muchos otros eventos de la interfaz de usuario.
4. Proporciona características de diseño avanzadas.
5. Contiene muchas propiedades específicas a la presentación visual, como **ForeColor**, **BackColor**, **Height** y **Width**.
6. Proporciona la seguridad y compatibilidad para subprocessos necesarias para que un control de formularios Windows Forms actúe como un control de Microsoft® ActiveX®.

Tipos de Controles

1. **CONTROLES COMPUESTOS**

Se trata de un grupo de controles de formularios WinForms dentro de un contenedor común. En algunas bibliografías puede llamarse control de usuario.

Desde la opción Proyecto del menú y un nuevo elemento puede agregar un control de usuario.



Veamos el siguiente código de ejemplo:

```
public partial class UserControl1 : UserControl
{
    public UserControl1()
    {
        InitializeComponent();
    }

    private void UserControl1_Load(object sender, EventArgs e)
    {
    }
}
```

2. CONTROLES AMPLIADOS

Un control ampliado es cuando queremos preservar la funcionalidad de un control existente pero además le agregamos nuevas funcionalidades.

```
public class cajaTextoNueva : System.Windows.Forms.TextBox {  
  
    protected override void OnPaint(System.Windows.Forms.PaintEventArgs e) {  
        base.OnPaint(e);  
    }  
}
```

REPORTES

Un **reporte** es un informe que organiza y exhibe la información contenida en una base de datos. Su función es aplicar un formato determinado a los datos para mostrarlos por medio de un diseño atractivo y que sea fácil de interpretar por los usuarios.

Los reportes se guardan como archivos de definición de informe del cliente (.rdlc), estos archivos se basan en el mismo esquema que los archivos de definición de reporte (.rdl) publicados en los servidores de informes de SQL Server Reporting Services, pero se guardan y se procesan de manera distinta a los archivos .rdl.

En tiempo de ejecución, los archivos .rdlc se procesan localmente, y los archivos .rdl se procesan remotamente

.NET tiene una librería:

```
using Microsoft.Reporting.WinForms;
```

El control ReportViewer admite un modo de procesamiento local que le permite ejecutar archivos de definición de informe de cliente (.rdlc) utilizando la capacidad de procesamiento integrada del control. Los informes de cliente que se ejecutan en modo de procesamiento local se pueden crear fácilmente en el proyecto de aplicación.

Para poder utilizar dicho control, tenemos que tener la opción de reportes en el menú de visual studio.

CHART

Los controles **Chart** permiten crear páginas ASP.NET o aplicaciones Windows Forms con gráficos simples, intuitivos y visualmente atractivos para análisis estadísticos complejos.

Características

- **Escalabilidad:** Soporta un número ilimitado de áreas de gráficos, títulos, leyendas y anotaciones.
- **Tipos de gráficos:** 35 tipos de gráficos distintos.
- **Personalización:** Manipulación de gráficos en tiempo real.

Propiedades

- **ChartAreas:** es el área donde se traza un gráfico. Se puede contener más de un gráfico por renderizado e incluso puede superponer tablas.
- **Series:** Son los datos que puede trazar en el área de su gráfico.
- **ChartType:** la propiedad de tipo de gráfico se encuentra bajo la propiedad Series y define cómo se mostrará la serie de datos en el gráfico.
- **Axes:** define propiedades para los ejes X e Y, como apariencia y títulos.
- **Palette:** define los colores establecidos para su gráfico.
- **Titles:** define el texto que se puede usar para describir un gráfico, un eje o cualquier otra parte del gráfico.
- **Legends:** define las leyendas que mostrarán la información de la serie de datos.
- **Labels:** define el texto que se puede mostrar cerca del eje, los puntos y las etiquetas personalizadas.

UNIDAD 5 - ESTRATEGIA DE DISEÑO DE ALGORITMOS

Itinerario 14

ESTRATEGIA DE DISEÑO

Se puede definir un **Algoritmo** como un 'Método para resolver un problema mediante una serie de pasos **precisos, definidos y finitos**'.

1) RECURSIVIDAD

La **recursividad** es una técnica fundamental en el diseño de algoritmos eficientes, que está basada en la solución de versiones más pequeñas del problema, para obtener la solución general del mismo. Una instancia del problema se soluciona según la solución de una o más instancias diferentes y más pequeñas que ella.

Es una herramienta poderosa que sirve para resolver cierto tipo de problemas reduciendo la complejidad y ocultando los detalles del problema. Esta herramienta consiste en que una función o procedimiento se llama a sí mismo.

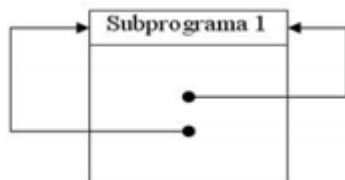
Una gran cantidad de algoritmos pueden ser descritos con mayor claridad en términos de recursividad, típicamente el resultado será que sus programas serán más pequeños.

La característica importante de la recursividad es que siempre existe un medio de salir de la definición, mediante la cual se termina el proceso recursivo.

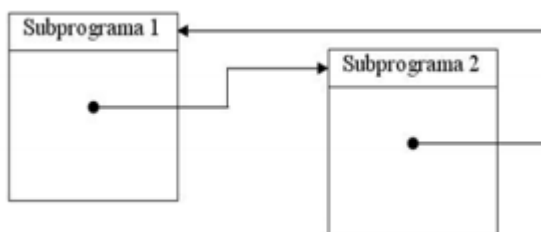
VENTAJAS	DESVENTAJAS
Puede resolver problemas complejos	Se puede llegar a un ciclo infinito
Solución más natural	Versión no recursiva más difícil de desarrollar
	Difícil de programar sin experiencia

Tipos de Recursividad

1. **Directa/Simple:** Un subprograma se llama a sí mismo una o más veces directamente.



2. **Indirecta/Mutua:** Un subprograma A llama a otro subprograma B y éste a su vez llama al subprograma A.



2) **DIVIDE PARA CONQUISTAR**

La técnica **Dividir para Conquistar (o Divide y Vencerás)** consiste en descomponer el caso que hay que resolver en subcasos más pequeños, resolver independientemente los subcasos y por último combinar las soluciones de los subcasos para obtener la solución del caso original.

Esta estructura obedece a una estrategia dividir-y-conquistar, en que se ejecuta tres pasos en cada nivel de la recursión:

- **Dividir:** Dividen el problema en varios subproblemas similares al problema original pero de menor tamaño;
- **Conquistar:** Resuelven recursivamente los subproblemas si los tamaños de los subproblemas son suficientemente pequeños, entonces resuelven los subproblemas de manera directa; y luego,
- **Combinar:** Combinan estas soluciones para crear una solución al problema original.

3) **PROGRAMACIÓN DINÁMICA**

Puede ocurrir que la división natural del problema conduzca a un gran número de sub ejemplares idénticos. Si se resuelve cada uno de ellos sin tener en cuenta las posibles repeticiones, resulta un algoritmo ineficiente; en cambio sí se resuelve cada ejemplar distinto una sola vez y se conserva el resultado, el algoritmo obtenido es mucho mejor.

Esta es la idea de la programación dinámica: no calcular dos veces lo mismo y utilizar normalmente una tabla de resultados que se va rellenando a medida que se resuelven los sub ejemplares.

La programación dinámica es un método ascendente. Se resuelven primero los sub ejemplares más pequeños y por tanto más simples. Combinando las soluciones se obtienen las soluciones de ejemplares sucesivamente más grandes hasta llegar al ejemplar original.

3) **ALGORITMO ÁVIDO**

Los algoritmos ávidos (Greedy Algorithms) son algoritmos que toman decisiones de corto alcance, basadas en información inmediatamente disponible, sin importar consecuencias futuras.

Suelen ser bastante simples y se emplean sobre todo para resolver problemas de optimización, como por ejemplo, encontrar la secuencia óptima para procesar un conjunto de tareas por un computador, hallar el camino mínimo de un grafo, etc.

Habitualmente, los elementos que intervienen son:

- un conjunto o lista de **candidatos** (tareas a procesar, vértices del grafo, etc.);
- un conjunto de **decisiones** ya tomadas (candidatos ya escogidos);
- una función que determina si un conjunto de candidatos es una solución al problema (aunque no tiene por qué ser la óptima);
- una **función** que determina si un conjunto es completable, es decir, si añadiendo a este conjunto nuevos candidatos es posible alcanzar una **solución** al problema, suponiendo que esta exista;
- una **función** de selección que escoge el candidato aún no seleccionado que es más **prometedor**;
- una **función objetivo** que da el valor/coste de una solución (tiempo total del proceso, la longitud del camino, etc.) y que es la que se pretende maximizar o minimizar;

Para resolver el problema de optimización hay que encontrar un conjunto de candidatos que optimicen la función objetivo. Los algoritmos voraces proceden por pasos.

Inicialmente el conjunto de candidatos es vacío. A continuación, en cada paso, se intenta añadir al conjunto el mejor candidato de los aún no escogidos, utilizando la función de selección. Si el conjunto resultante no es completable, se rechaza el candidato y no se le vuelve a considerar en el futuro. En caso contrario, se incorpora al conjunto de candidatos escogidos y permanece siempre en él. Tras cada incorporación se comprueba si el conjunto resultante es una solución del problema. Un algoritmo voraz es correcto si la solución así encontrada es siempre óptima.

EFICIENCIAS

Principio de Invarianza

Dado un algoritmo y dos implementaciones suyas I_1 e I_2 , que tardan $T_1(n)$ y $T_2(n)$ segundos respectivamente, el **Principio de Invarianza afirma que** existe una constante real $c > 0$ y un número natural n_0 tales que para todo $n \geq n_0$ se verifica que $T_1(n) \leq cT_2(n)$.

Es decir, **el tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa.**

Esto significa que el tiempo para resolver un problema mediante un algoritmo depende de la naturaleza del algoritmo y no de la implementación del algoritmo.

Cuando se quiere comparar la eficiencia temporal de dos algoritmos, tiene mayor influencia el tipo de función que la constante c.

Tiempo de ejecución

Se le llama **tiempo de ejecución**, no al tiempo físico, sino al **número de operaciones elementales que se llevan a cabo en el algoritmo**.

El tiempo que requiere un algoritmo para dar una respuesta, se divide generalmente en 3 casos:

- **Peor Caso:** caso más extremo, donde se considera el tiempo máximo para solucionar un problema.
- **Caso promedio:** caso en el cual, bajo ciertas restricciones, se realiza un análisis del algoritmo
- **Mejor caso:** caso ideal en el cual el algoritmo tomará el menor tiempo para dar una respuesta

Para medir $T(n)$ usamos el número de operaciones elementales, las mismas pueden ser:

- Operación aritmética.
- Asignación a una variable.
- Llamada a una función.
- Retorno de una función.
- Comparaciones lógicas (con salto).
- Acceso a una estructura (arreglo, matriz, lista ligada...).

COMPLEJIDAD ALGORÍTMICA

Medidas asintóticas

Las cotas de complejidad, también llamadas **medidas asintóticas** sirven para clasificar funciones de tal forma que podamos compararlas. Las medidas asintóticas permiten analizar qué tan rápido crece el tiempo de ejecución de un algoritmo cuando crece el tamaño de los datos de entrada, sin importar el lenguaje en el que esté implementado ni el tipo de máquina en la que se ejecute.

Existen diversas notaciones asintóticas para medir la complejidad, las **tres cotas de complejidad más comunes son**: la notación O (o mayúscula), la notación Ω (omega mayúscula) y la notación θ (theta mayúscula) y todas se basan en el peor caso.

Cota superior asintótica: Notación O (o mayúscula): Dada una función f , se estudian todas aquellas funciones g que a lo sumo crecen tan deprisa como f . Al conjunto de tales funciones se le llama cota superior de f y lo denominamos $O(f)$. Conociendo la cota superior de un algoritmo se puede asegurar que, en ningún caso, el tiempo empleado será de un orden superior al de la cota.

Cota inferior asintótica: Notación Ω (omega mayúscula): Dada una función f , se quieren estudiar aquellas funciones g que a lo sumo crecen tan lentamente como f . Al conjunto de tales funciones se le llama cota inferior de f y se denominan $\Omega(f)$. Conociendo la cota inferior de un algoritmo se puede asegurar que, en ningún caso, el tiempo empleado será de un orden inferior al de la cota.

Orden exacto o cota ajustada asintótica: Notación θ (theta mayúscula) Como última cota asintótica, están los conjuntos de funciones que crecen asintóticamente de la misma forma

Ejemplo: ¿Cuál es la complejidad de multiplicar dos enteros?

- Depende de cual sea la medida del tamaño de la entrada.
- Podrá considerarse que todos los enteros tienen tamaño $O(1)$, pero eso no será útil para comparar este tipo de algoritmos.
- En este caso, conviene pensar que la medida es el logaritmo del número.
- Si por el contrario estuviéramos analizando algoritmos que ordenan arreglos de enteros, lo que importa no son los enteros en sí, sino cuántos tengamos.
- Entonces, para ese problema, la medida va a decir que todos los enteros miden lo mismo

Programación paralela y secuencial

La **programación secuencial** es cuando **una tarea va después de otra**. Es un proceso lento en el que, **si una tarea se retrasa, el sistema completo debe esperar**. La ventaja es que es **fácil de entender y de implementar**.

Mientras que **programación paralela** es el uso de **múltiples recursos computacionales para resolver un problema**.

En el sentido más simple, la programación paralela es el uso simultáneo de múltiples recursos computacionales para resolver un problema computacional:

- Un problema se divide en partes discretas que se pueden resolver simultáneamente.
- Cada parte se descompone en una serie de instrucciones
- Las instrucciones de cada parte se ejecutan simultáneamente en diferentes procesadores
- Se emplea un mecanismo global de control/coordinación.

Algoritmo Distribuido

Un algoritmo distribuido es un tipo específico de algoritmo que se utiliza en sistemas distribuidos y que ha sido diseñado para aprovechar las características de este tipo de sistemas.

Propiedades de algoritmos distribuidos:

- La información relevante se distribuye entre varias máquinas.
- Se toman decisiones sólo en base a la información local.
- Debe evitarse un punto único de fallo.
- No existe un reloj común.

FASES DE RESOLUCIÓN DE PROBLEMAS

La **resolución de un problema** consiste en el proceso que a partir de la descripción de un problema, expresado habitualmente en lenguaje natural y en términos propios del dominio del problema, permite desarrollar un programa que resuelva dicho problema.

Este proceso exige los siguientes pasos:

1. **Análisis del problema:** El problema se analiza teniendo presente la especificación de los requisitos dados por el cliente de la empresa o por otra persona que encarga el programa.
2. **Diseño o desarrollo de un algoritmo:** una vez analizado el problema, se diseña una solución que conducirá a un algoritmo que resuelva el problema
3. **Codificación** (implementación): Esta etapa consiste en transcribir o adaptar el algoritmo a un lenguaje de programación, se tendrá que adaptar todos los pasos diseñados en el algoritmo con sentencias y sintaxis propias del lenguaje.
4. **Ejecución, verificación y depuración:** el programa se ejecuta, se comprueba rigurosamente y se elimina todos los errores (denominados “bugs”, en inglés) que puedan aparecer
5. **Mantenimiento:** El programa se actualiza y modifica, cada vez que sea necesario, de modo que se cumplan todas las necesidades de cambio de sus usuarios.