Formato de Código

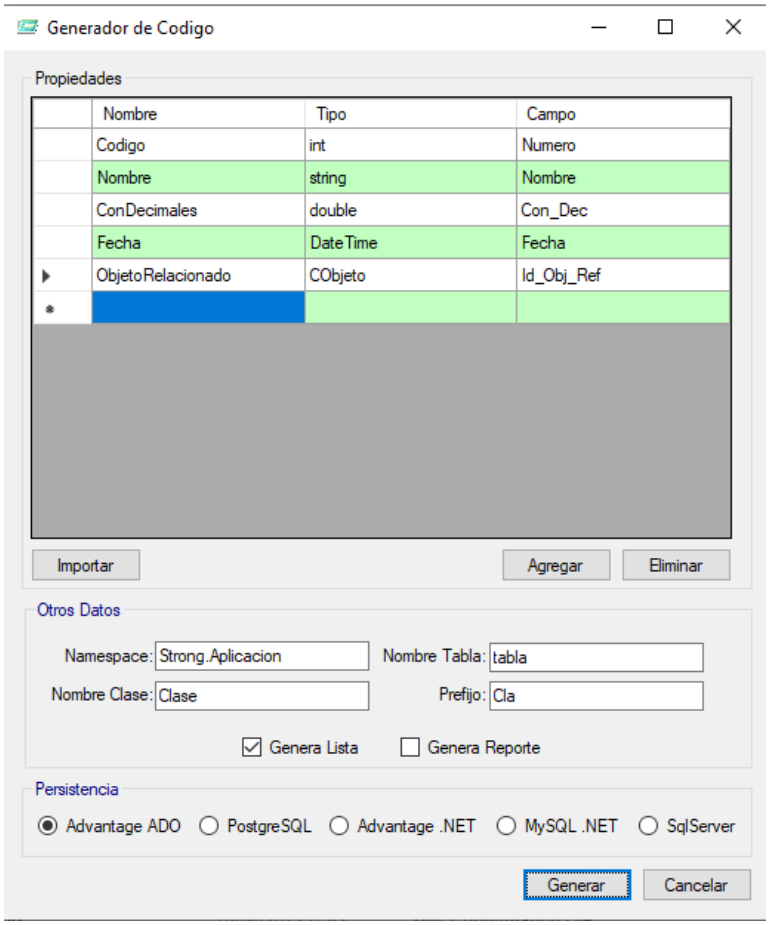
Con el fin de unificar el formato del código que escribimos, facilitando la lectura por parte de todos los integrantes del equipo, definimos algunos lineamientos que deben mantenerse para todas las aplicaciones que generemos, dentro de la medida de lo posible.

Para facilitar la codificación de lo que puede armarse automáticamente, disponemos de un generador que, dada una tabla, permite generar la clase que implementa su lógica y el acceso a datos para ésta.

Generador de Código

En el mismo cargamos todas las columnas de la tabla que vamos a manejar (puede importarse directamente, pero luego hay que realizar algunos ajustes).

Podemos usar tipos de datos primitivos u objetos. Para los casos que tenemos objetos referenciados luego tendremos que modificar el código generado.



Esto va a generar tres archivos:

- CClase – Clase que representa la lógica
- PClase – Persistencia para la clase antes generada
- LClase – Manejo de listas de objetos de la clase antes generada

La clase generada (capa lógica)

Se implementa una clase que contiene una propiedad para cada columna de la tabla asignada a un tipo primitivo, y objetos para representar las referencias.

Cuando tenemos objetos, debemos agregar manualmente el método *Instanciar* y corregir el *Inicializar* para que llame a las inicializaciones de los objetos referenciados.

He aquí el código generado:

```
using Strong.Aplicacion.Persistencia.Adv;

namespace Strong.Aplicacion.Dominio {

      public class CClase {

           #region Propiedades

           public int Codigo { get; set; }
           public string Nombre { get; set; }
           public double ConDecimales { get; set; }
           public DateTime Fecha { get; set; }
           public CObjeto ObjetoRelacionado { get; set; }

           #endregion


           #region Construcción

           public CClase() {
                Instanciar();
                Inicializar();
           }

           public void Instanciar() {
                ObjetoRelacionado = new CObjeto();
           }

           public void Inicializar() {
                Codigo = 0;
                Nombre = string.Empty;
                ConDecimales = 0;
                Fecha = DateTime.Now;
                ObjetoRelacionado.Inicializar();
           }

           #endregion

           #region Persistencia

           public bool Agregar() {
                PClase pers = new PClase();
                return new pers.Agregar(this);
           }

           public void Cargar(Recordset rs) {
                PClase pers = new PClase();
                pers.Cargar(this, rs);
           }

           public void Eliminar() {
                PClase pers = new PClase();
                pers.Eliminar(this);
           }

           public bool Existe()       {
                PClase pers = new PClase();
                return pers.Existe(this);
           }

           public bool Modificar() {
                PClase pers = new PClase();
                return pers.Modificar(this);
           }
```

```
          public bool Recuperar() {
               PClase pers = new PClase();
               return pers.Recuperar(this);
          }

          #endregion


          #region Strings

          public override string ToString() {
               return codigo + " " + nombre;
          }

          #endregion


          #region Públicos

          // métodos públicos de la clase, si no hay se quita la región

          #endregion

          #region Privados

          // métodos privados de la clase, si no hay se quita la región

          #endregion

     }
}
```

Dentro de las regiones de métodos públicos y privados, agregamos los demás métodos que hagan falta. Si no tiene ninguno, quitamos la región.

Persistencia

Para cada clase de nuestra capa lógica, tendremos una clase vinculada a ella que se encarga de resolver su persistencia, brindando métodos para leer, insertar, modificar y eliminar. Además de los *recuperar* y *modificar* generales (que cubren todos los campos) pueden existir específicos, que afecten solo algunos de sus campos, para operaciones que se ejecutan con frecuencia y no afectan todos los campos.

Considerando la clase que venimos viendo, esta sería su persistencia:

```
using ADODB;
using ConexionAdv;
using Strong.Aplicacion.Dominio;

namespace Strong.Aplicacion.Persistencia.Adv {

   public class PClase {

        #region Atributos

        private Connection conexion;

        #endregion

        #region Construcción

        public PClase() {
             conexion = CConexion.Instancia().Conexion();
        }

        #endregion


        //Base de datos a objeto
```

```
#region Campos

public string Campos() {
     PObjeto pObj = new PObjeto();
     return " tabla.Numero AS ClaCodigo," +
            " tabla.Nombre AS ClaNombre," +
            " tabla.Con Dec AS ClaConDecimales," +
            " tabla.Fecha AS ClaFecha," +
            pObj.Campos();
}

#endregion

#region Cargar

public void Cargar(CClase obj, Recordset rs) {
     obj.Inicializar();
     if (!rs.EOF) {
        if (!Convert.IsDBNull(rs.Fields["ClaCodigo"].Value))
               obj.Codigo = Convert.ToInt32(rs.Fields["ClaCodigo"].Value);
        if (!Convert.IsDBNull(rs.Fields["ClaNombre"].Value))
               obj.Nombre = Convert.ToString(rs.Fields["ClaNombre"].Value);
        if (!Convert.IsDBNull(rs.Fields["ClaConDecimales"].Value))
               obj.ConDecimales = Convert.ToDouble
                                          (rs.Fields["ClaConDecimales"].Value);
        if (!Convert.IsDBNull(rs.Fields["ClaFecha"].Value))
               obj.Fecha = Convert.ToDateTime(rs.Fields["ClaFecha"].Value);

        obj.ObjetoRelacionado.Cargar(rs);
     }
}

#endregion

#region Recuperar

public bool Existe(CClase obj) {
     Recordset rs = new Recordset();
     string sql;
     bool resultado;

     sql = "SELECT * " +
          "FROM " + Tabla() +
          "WHERE " + Condicion(obj);

     rs.Open(sql, conexion);

     resultado = !rs.EOF;
     rs.Close();

     return resultado;
}

public bool Recuperar(CClase obj) {
     Recordset rs = new Recordset();
     string sql;
     bool resultado;

     sql = "SELECT " + Campos() +
          "FROM " + Tablas() +
          "WHERE " + Condicion(obj);

     rs.Open(sql, conexion);

     Cargar(obj, rs);
     resultado = !rs.EOF;
     rs.Close();

     return resultado;
}

#endregion


//Objeto a Base de Datos

#region Condiciones
```

```
public string Condicion(CClase obj)  {
     return " tabla.Numero = " + obj.Codigo + " ";
}

#endregion

#region DML

public bool Agregar(CClase obj) {
     string sql;
     object registros;

     sql = "INSERT INTO " + Tabla() +
          "(" + AgregarVariables() + ")" +
          " VALUES " +
          "(" + AgregarValores(obj) + ")";

     conexion.Execute(sql, out registros, 0);

     return ((int)registros > 0);
}

public void Eliminar(CClase obj) {
     string sql;
     object registros;

     sql = "DELETE FROM " + Tabla() + " " +
          "WHERE " + Condicion(obj);

     conexion.Execute(sql, out registros, 0);
}

public bool Modificar(CClase obj) {
     string sql;
     object registros;

     sql = "UPDATE " + Tabla() + " " +
               "SET " + ModificarValores(obj) + " " +
            "WHERE " + Condicion(obj);

     conexion.Execute(sql, out registros, 0);

     return ((int)registros > 0);
}

#endregion

#region Edición

public string AgregarValores(CClase obj) {
     return obj.Codigo + ", " +
           CFuncionesBd.StringBd(obj.Nombre) + ", " +
           CFuncionesBd.DoubleBd(obj.ConDecimales) + ", " +
           CFuncionesBd.DateTimeBd(obj.Fecha) + ", " +
           obj.ObjetoRelacionado.Codigo;
}

public string AgregarVariables() {
     return "Numero, " +
            "Nombre, " +
            "Con_Dec, " +
            "Fecha, " +
            "Id Obj Ref ";
}

public string ModificarValores(CClase obj)  {
     return "Nombre = " + CFuncionesBd.StringBd(obj.Nombre) + ", " +
            "Con Dec = " + CFuncionesBd.DoubleBd(obj.ConDecimales) + ", " +
            "Fecha = " + CFuncionesBd.DateTimeBd(obj.Fecha) + ", " +
            "Id_Obj_Ref = " + obj.ObjetoRelacionado.Codigo;
}

#endregion


//Tablas
```

```
        #region Orden

        public string Orden() {
             return " tabla.Numero ";
        }

        #endregion

        #region Tablas

        public string Tabla() {
             return " tabla ";
        }

        public string Tablas() {
             return " tabla" +
                    " LEFT OUTER JOIN tabla_rel ON (table.Id_Obj_Ref = table_rel.Id) ";
        }

        #endregion

    }
}
```

Cuando concatenamos parámetros que son de tipo String, double, decimal o DateTime, tenemos métodos que aseguran que se haga de manera segura. Estos métodos están en la biblioteca CFuncionesBd.

En el caso que la clase no tenga otras clases relacionadas, la persistencia se puede simplificar, eliminando la función de tablas, ya que no necesitará Joins.

Si hubiesen recuperar específicos que necesiten menos información, pueden utilizar su propia combinación de tablas, con sus métodos específicos.

En líneas generales, cada método de Campos debe corresponderse con su método Cargar, y eventualmente con su método de Tablas (este último puede compartirse por varias parejas Campos-Cargar).

En la actualidad estamos trabajando en mejorar la forma de obtener los parámetros desde los Recordsets, para evitar el Convert en cada acceso, pero eso lo pondremos en práctica para la próxima versión del estándar. Con este nuevo modo, podremos remplazar nuestro *Cargar* por algo como este:

```
public void Cargar(CClase obj, Recordset rs) {
    obj.Inicializar();
    if (!rs.EOF) {
         obj.Codigo = rs.GetInt("ClaCodigo");
         obj.Nombre = rs.GetString("ClaNombre");
         obj.ConDecimales = rs.GetDouble("ClaConDecimales");
         obj.Fecha = rs.GetDateTime("ClaFecha");

         obj.ObjetoRelacionado.Cargar(rs);
    }
}
```

Este manejo se basa en la creación de los *Get* como métodos de extensión del tipo de datos Recordset.

Manejo de Listas

Todo lo que refiere a obtener listas de objetos de la clase que estamos creando, se debe resolver en su “L”. Esta clase es parte de la persistencia y puede no existir si no hay necesidad de manejar listas.

Siempre las vamos a utilizar para resolver consultas que obtengan como resultado colecciones de objetos de esta clase. Puede ser un “recuperar todos” u “obtener aquellos que cumplen determinada condición”.

El código que presentamos a continuación se divide en dos partes, una de ellas es enteramente opcional (ver comentarios).

```
using ADODB;
using ConexionAdv;
using Strong.Aplicacion.Dominio;

namespace Strong.Aplicacion.Persistencia.Adv {

   public class LClase {

        #region Atributos

        Connection conexion;

        #endregion

        #region Constructor

        public LClase() {
              conexion = CConexion.Instancia().Conexion();
        }

        #endregion

        #region Lista

        // este código busca en listas de objetos que tienen propiedades Codigo y Nombre
        // si la clase lo tiene, la dejamos tal cual, si no tiene, se quita la región

        public List<CClase> BuscarCodigo(int codigo, List<CClase> lista){
              List<CClase> resultados = new List<CClase>();
              string strCodigo, strCodigoActual;

              strCodigo = codigo.ToString().Trim();

              foreach (CClase actual in lista) {
                   strCodigoActual = actual.Codigo.ToString().Trim();
                   if (strCodigoActual.IndexOf(strCodigo) != -1) {
                        resultados.Add(actual);
                   }
              }

              return resultados;
        }

        public List<CClase> BuscarNombre(string nombre, List<CClase> lista) {
              List<CClase> resultados = new List<CClase>();
              string strNombre, strNombreActual;

              strNombre = nombre.ToUpper().Trim();

              foreach (CClase actual in lista) {
                   strNombreActual = actual.Nombre.ToUpper().Trim();

                   if (strNombreActual.IndexOf(strNombre) != -1) {
                        resultados.Add(actual);
                   }
              }

              return resultados;
        }

        #endregion
```

```
        #region Recuperar

        public List<CClase> RecuperarTodos() {
              Recordset rs = new Recordset();
              List<CClase> lista = new List<CClase>();
              PClase pers = new PClase();
              CClase obj;
              string sql;

              sql = "SELECT " + pers.Campos() +
                    "FROM " + pers.Tablas() +
                    "ORDER BY " + pers.Orden();

              rs.Open(sql, conexion);

              while (!rs.EOF) {
                   obj = new CClase();
                   pers.Cargar(obj, rs);
                   lista.Add(obj);
                   rs.MoveNext();
              }

              rs.Close();

              return lista;
        }

        // aquí va cualquier otro Recuperar que traiga listas de estos objetos
        // que cumplan determinadas condiciones

        #endregion

    }
}
```

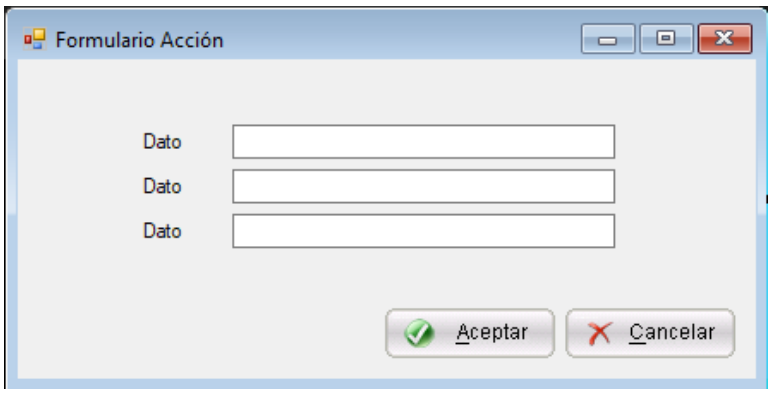
La capa de presentación

Si bien para la capa de presentación no tenemos un generador de código, lo que sí tenemos es una biblioteca con muchos controles de usuario, prontos para ser utilizados. En ella hay, además de controles multipropósito, paneles que resuelven la selección de diferentes códigos, que suelen utilizarse en muchas de las ventanas de nuestros sistemas.

Como líneas generales en cuanto al diseño, veremos tres tipos de formularios, con las características que deben cumplir siempre que se estén diseñando nuevas funcionalidades.

Acciones

Los formularios que ejecutan una determinada acción en líneas generales tienen el siguiente formato.



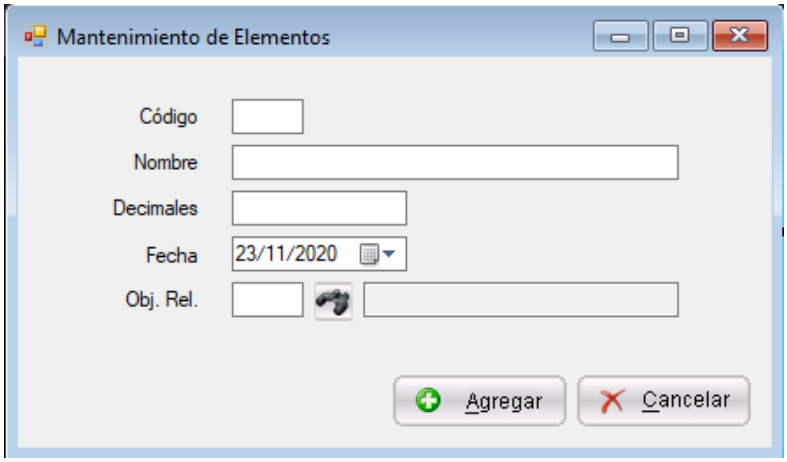
Presentan campos para manejar los datos necesarios para la acción, y botones de Aceptar y Cancelar. Los mismos, a pesar de ser imágenes, tienen cargados los textos “&a” y “&c” para funcionar con las teclas Alt.

El botón “Aceptar” ejecuta las validaciones necesarias, y en caso que las cumpla, pide una confirmación, realiza la acción y se cierra.

El botón “Cancelar” no ejecuta validaciones (propiedad *CausesValidation*) y cierra el formulario. Además, es el *CancelButton* del formulario.

Altas, Bajas y Modificaciones (CRUDs)

Las pantallas de mantenimientos tienen una apariencia como la que se muestra a continuación.



El funcionamiento comienza parado en el código. En la validación del mismo, busca el elemento identificado por dicho código, si no existe lo trata como un alta, y si existe como un mantenimiento.

Carga de datos

En el objeto relacionado, cargamos el código y se valida que se indique uno existente. Con el botón del prismático (o con F9) podemos buscarlo en una lista. En el *TextChange* se limpia el nombre mostrado, en el *Validating* se busca el código digitado y si existe se muestra. Si no existe, se cancela la validación. Todos los datos que puedan requerir validaciones, se controlan en la medida que se van ingresando.

Altas

Cuando presionamos el botón “Agregar” se efectúan las validaciones necesarias, se verifica que estén completos todos los datos requeridos, se pide confirmación y posteriormente se realiza el alta. Acto seguido, se resetea el formulario, quedando nuevamente parado en el código (con todos los campos “limpios”).

Modificaciones y Bajas

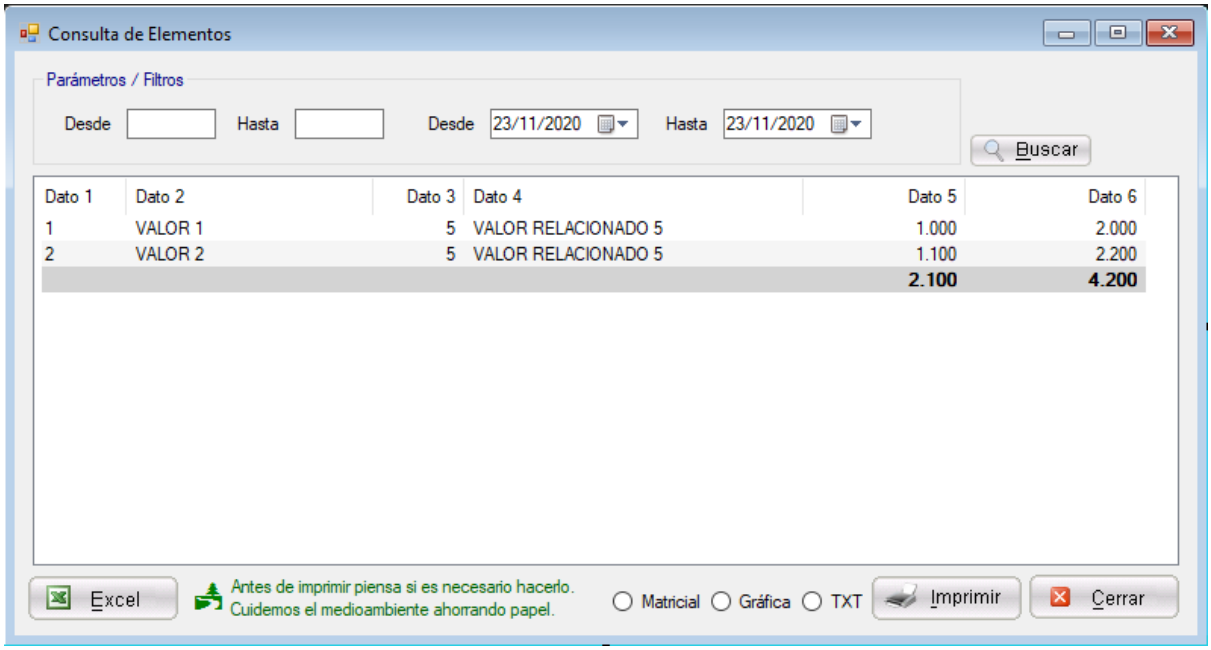
Cuando en la validación del código, elegimos uno que ya existe, se oculta el botón “Agregar” y se muestran los botones “Modificar” y “Eliminar”.

El botón “Eliminar” realiza las validaciones correspondientes para determinar si se puede eliminar, en caso que las cumpla, pide confirmación y lo elimina. Posteriormente resetea el formulario y vuelve a comenzar.

En el clic del botón “Modificar” se efectúan las validaciones necesarias, se verifica que estén completos todos los datos requeridos, se pide confirmación y posteriormente se realiza la modificación. Acto seguido, se resetea el formulario, quedando nuevamente parado en el código (con todos los campos “limpios”).

Consultas / Listados

Las consultas pueden ser de lo más variadas, pero todas tienen cierta similitud dentro del formato del formulario, los cuales en líneas generales lucen como el siguiente.



En la parte superior tenemos todos los filtros que podemos aplicar para buscar la información. Cambiando cualquiera de ellos limpiamos los valores que se muestran en la lista de resultados.

Los botones al pie, desde “Excel” a “Imprimir” están dentro de un panel que tiene las operaciones implementadas para cualquier lista. Este se debe alimentar con la información indicándole de qué lista proviene.

El botón “Buscar” ejecuta la consulta y carga los resultados en la lista. En este punto se alimenta el panel antes mencionado con la lista, una vez que ésta ya haya sido cargada.

Siempre debemos cuidar que la información en la lista quepa en la pantalla. Siempre en la medida de lo posible, debemos evitar el scroll horizontal.

En contadas ocasiones, puede que la lista en pantalla y lo que queremos imprimir tenga ciertas diferencias. Esto puede ser porque hay mucha información en pantalla y no se puede imprimir directamente. En estos casos, hacemos el método “imprimir” manualmente. Para ello, existe una clase llamada *CReporte*, que facilita la impresión en distintos tipos de impresoras.

Contenido