UTN
Facultad Regional Córdoba

TUP

Tecnicatura Universitaria en Programación

PROGRAMACIÓN II

Unidad Temática N°1:
Gestión Maestro-Detalle

Material Teórico
1° Año – 2° Cuatrimestre

Índice

GESTION MAESTRO-DETALLE 2

Introducción 2
Repaso de ADO.NET 2

PROCEDIMIENTOS ALMACENADOS CON ADO.NET 4

Parámetros de salida 6

ABMC MAESTRO-DETALLE 8

Caso de estudio: Presupuestos de Carpintería 8
Creando la solución 9
Nuevo Presupuesto 11
Eventos y código C# 11

MANEJO DE TRANSACCIONES 18

Refactorizando el método Confirmar() 19

REPORTES EN C# 21

Pasos para Generar un Reporte 22
Listados utilizando un TableAdapter 27
Reportes utilizando una Tabla propia 35

BIBLIOGRAFÍA 41

ANEXO: INSTALACIÓN DE COMPLEMENTOS VISUAL STUDIO 42

Instalación de Microsoft RDLC Report Designer 42
Instalación de Paquete NuGet para el control ReportViewer 42

GESTION MAESTRO-DETALLE

Introducción

Son comunes las aplicaciones en las cuales se requiere mostrar y actualizar los registros de una tabla junto con registros asociados de otra tabla. Este tipo de relación entre registros comúnmente se conoce como **Maestro - Detalle**. Son ejemplos típicos: Factura-Detalle de Factura, Orden de Compra – Artículos que pertenecen a cada Orden, Departamento - Empleados o bien un Informe de gastos – Partidas de gastos. Desde el punto de vista de la POO estos modelos corresponden a relaciones de Agregación entre las entidades de dominio.

El objetivo de esta unidad es reforzar los conceptos de acceso a datos mediante el modelo ADO.NET visto en Programación I, para gestionar las entidades de una base de datos relacional vinculadas mediante una relación Maestro-Detalle. Se incorpora el uso de procedimientos para encapsular las sentencias SQL necesarias y el manejo transaccional subyacente.

Por último, tomando como base este modelo, se analizarán las herramientas provistas por C# para la generación de reportes y salidas de información.

Repaso de ADO.NET

La secuencia de pasos a seguir para conectarnos a un motor SQL Server, acceder a nuestra BD y ejecutar comandos SQL se puede sintetizar en el siguiente esquema:

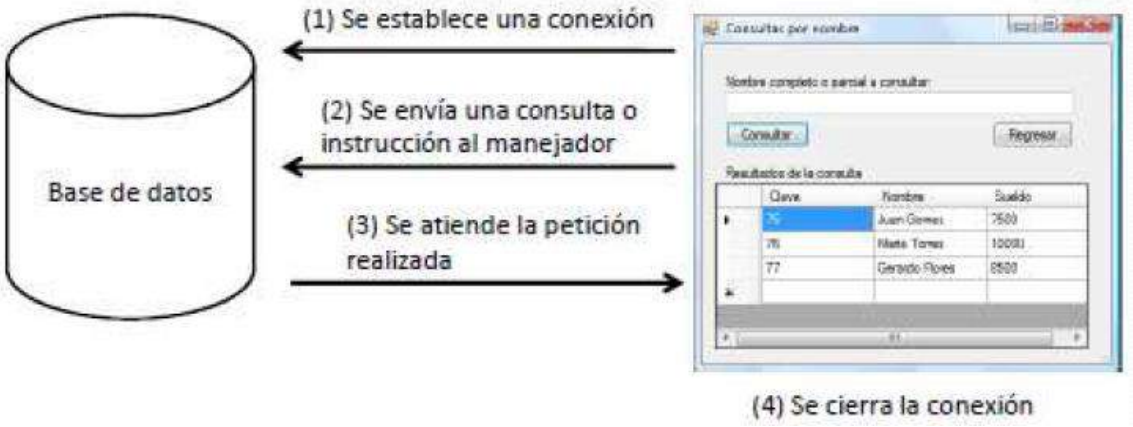

Imagen 1: Elaboración propia

Dónde los componentes que intervienen son:

- **Connection:**
  - Representa una conexión a la BD.
  - Permite abrir y cerrar la conexión a la BD.

- **Command:**
  - Representa una vía para representar sentencias SQL a la BD: Select, Insert, Delete, Update o bien la llamada a un Procedimiento Almacenado.
- **DataReader:**
  - Almacén temporal de datos, de sólo lectura y sólo hacia adelante.

La conexión se debe abrir y cerrar explícitamente dentro del programa. Siempre consume recursos hasta que es cerrada.

Otra alternativa es trabajar de manera **desconectado.** Una característica interesante de ADO.NET es la posibilidad de trabajar localmente con los datos en memoria para luego sincronizar y actualizar nuestra BD a posterior. Los pasos pueden visualizarse en el siguiente gráfico:

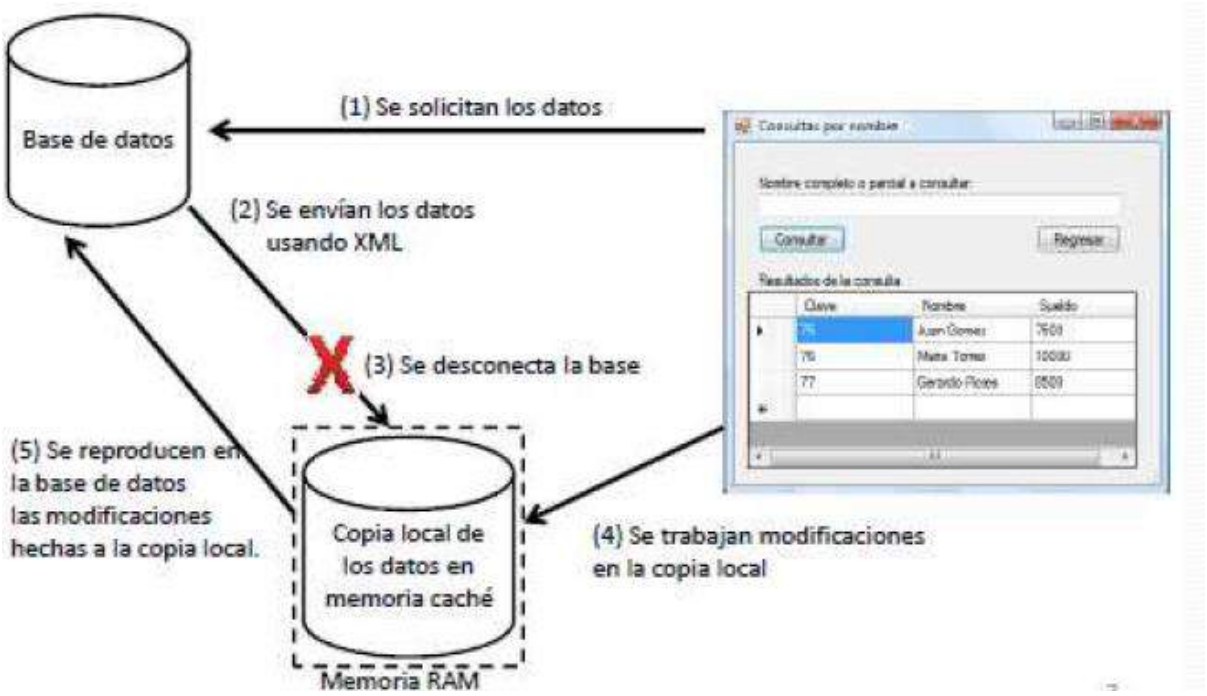

Imagen 2: Elaboración Propia

Dónde los objetos del componente que intervienen son:

- **DataAdapter:**
  - Conecta el programa con la BD, realiza consultas, llena los DataSet y sincroniza los cambios en la BD. Permite abrir y cerrar la conexión a la BD.
- **DataSet:**

- Es una “copia en memoria local” de una porción de la BD.
- Se encuentra en la memoria del cliente.
- Compatible con las BD relacionales (almacena datos en forma de tablas).

En este caso el objeto DataAdapter se conecta a la base de datos y realiza la consulta, pero no mantiene una conexión activa con la base de datos.

La arquitectura general de los objetos de ADO .NET puede resumirse como:

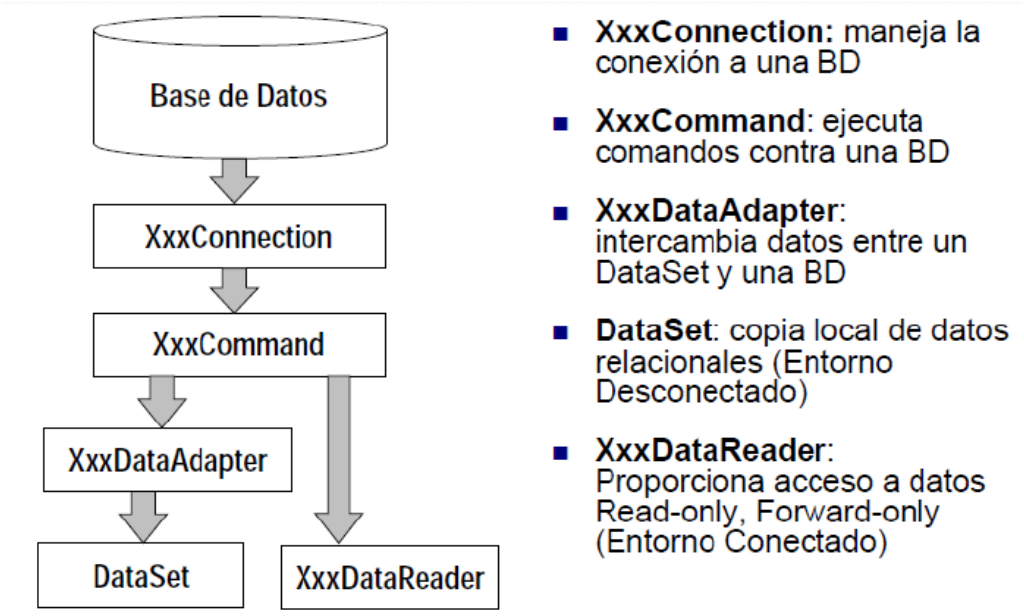

Imagen 3: Elaboración Propia

En síntesis, tenemos dos maneras de interactuar con nuestra BD, manteniendo la conexión activa hasta que se obtienen y/o actualizan los datos o de manera local mediante un conjunto de datos en memoria. De ambos mecanismos nos centraremos en el primero dejando esta segunda alternativa como inquietud para análisis y desarrollo por parte de los estudiantes.

PROCEDIMIENTOS ALMACENADOS CON ADO.NET

Un procedimiento almacenado (Store Procedure o SP en inglés) es un conjunto de sentencias SQL que aceptan y/o retornan cero o más variables, comúnmente llamados parámetros. Para nuestro propósito, los SP nos permitirán escribir todas las sentencias SQL necesarias para registrar, actualizar, eliminar y/o consultar las entidades gestionadas por la aplicación, eliminando todo código SQL de nuestro proyecto. Esta forma de incluir la lógica de la aplicación en la base de

datos utilizando procedimientos almacenados tiene como principal ventaja la simplificación del mantenimiento de nuestros programas.

Para consumir un SP desde C# utilizando los objetos de ADO necesitamos crear un objeto **Command** (en nuestro caso SQLCommand) y configurarlo para ejecutar correctamente el procedimiento. Para ello es necesario:

1. Obtener una conexión mediante un objeto **SQLConnection (SqlConnection cnn)**
2. Crear un objeto **SQLCommand** y asociarlo con el objeto conexión (**SQLCommand cmd**)
3. Asignar las propiedades Text y Type al comando:
    a. **cmd**.Connection = cnn;
    b. **cmd**.CommandText = “NOMBRE_DEL_PROCEDIMIENTO”;
    c. **cmd**.CommandType = CommandType.StoredProcedure;
    d. Establecer el/los parámetros del procedimiento (opcional):

    cmd.Parameters.AddWithValue("@param1",valor1); siendo valor1 una variable de nuestro programa y que necesitamos asignar al parámetro @param1 definido en el cuerpo del SP. La propiedad Parameters es una colección que almacena los objetos SQLParameter que serán enviados al ejecutarlo. El método AddWithValue(key, value) permite establecer para un parámetro su valor sin necesidad de definir un tipo específico. Será nuestra responsabilidad enviar los valores correctos según la definición del procedimiento en la base de datos.

    Otra forma es crear un objeto SQLParameter, setearle sus propiedades y luego agregarlo a la colección **Parameters.** Tal como lo muestra el siguiente método:

```
private static void AddSqlParameter(SqlCommand command)
{
        SqlParameter parameter = new SqlParameter();
        parameter.ParameterName = "@Description";
        parameter.IsNullable = true;
        parameter.SqlDbType = SqlDbType.VarChar;
        command.Parameters.Add(parameter);
```

```
        }
```

e. Executar el comando:

cmd.ExecuteNonQuery();

Parámetros de salida

Otro aspecto a considerar son los parámetros de retorno (o salida) que pudiera devolver el procedimiento. En la mayoría de los casos o no devuelve ningún valor o devuelve el resultado de una consulta. Por ejemplo, dado el siguiente Store Procedure escrito en SQLServer:

```
CREATE PROCEDURE [dbo].[SP_CONSULTAR_MODELOS]
@nombre varchar(40)
AS
BEGIN
SELECT * FROM T_MODELOS t WHERE t.nombre = @nombre;
END
Para ejecutarlo usamos programamos el siguiente método:
public void EjecutarSP(string nombreSP, Connection cnn)
    {
        try
        {
                SqlCommand cmd = new SqlCommand(nombreSP, cnn);
                 cmd.CommandType = CommandType.StoredProcedure;
                 cmd.Parameters.AddWithValue("@nombre","marca 1");
                 DataTable table = new DataTable();
                 table.Load(cmd.ExecuteReader());
         //procesar el resultado obtenido
          }
              catch (Exception ex)
        {
         throw new Exception(" Error al ejecutar procedimiento almacenado
", ex);
        }
    }
```

Suponiendo que tenemos una conexión creada y abierta a la base de datos, entonces podemos llamar al método anterior:

```
EjecutarSP ("SP_CONSULTAR_MODELOS",cnn);
```

Como vemos permite obtener como resultado la consulta de todos los modelos de la tabla T_MODELOS filtrada por nombre, siendo este último el único parámetro de entrada del procedimiento.

Veamos ahora un segundo ejemplo en el que insertamos una fila en la tabla T_MODELOS y obtenemos un parámetro de salida que representa el último ID generado para su clave primaria de tipo IDENTITY (Autoincremental):

```
CREATE PROCEDURE SP_INSERTAR_MODELO
    @nombre varchar(40),
    @NewId int OUTPUT
AS
BEGIN
    INSERT INTO T_MODELOS (nombre)
    VALUES (@nombre);
    --Asignamos el valor del último ID autogenerado (obtenido --
    --mediante la función SCOPE_IDENTITY() de SQLServer)
    SET @NewId = SCOPE_IDENTITY();
END
```

Para ejecutarlo refactorizamos el método anterior:

```
public void EjecutarSP(string nombreSP, Connection cnn)
    {
        try
        {
                 SqlCommand cmd = new SqlCommand(nombreSP, cnn);
                cmd.CommandType = CommandType.StoredProcedure;
                //parámetro de entrada:
                cmd.Parameters.AddWithValue("@nombre","marca 1");
                //parámetro de salida:
                SqlParameter param = new SqlParameter("@NewId", SqlDbType.Int);
                param.Direction = ParameterDirection.Output;
                cmd.Parameters.Add(param);
                //ejecutamos el comando
                cmd.ExecuteNonQuery();
```

```
                    //Obtenemos el valor del parámetro de salida:
                    int ultimoID = Convert.ToInt32(param.Value);
                }
                catch (Exception ex)
                {
                 throw new Exception(" Error al ejecutar procedimiento almacenado ", ex);
                }
            }
        //llamamos al método EjecutarSP:
        EjecutarSP ("SP_INSERTAR_MODELO",cnn);
```

En este caso el procedimiento tiene 2 parámetros: uno de entrada y otro de salida con el que obtenemos el valor del último ID generado para la columna ID_MODELO de la tabla T_MODELOS luego de ejecutar la sentencia **Insert**.

ABMC MAESTRO-DETALLE

Una relación maestro-detalle es una asociación entre dos bloques de datos que refleja una relación de **clave primaria - clave foránea** entre las tablas de Base de Datos en las que se basan los dos bloques de datos. El bloque de datos maestro se basa en la tabla con una clave primaria y el bloque de datos de detalle se basa en la tabla con una clave foránea. Una relación maestro-detalle equivale a la relación uno-a-muchos en el diagrama de relación de entidad.

A continuación, vamos a desarrollar un caso de estudio que ejemplifique la relación Maestro-Detalle mediante la creación de proyecto Aplicación de Windows Forms.

Caso de estudio: Presupuestos de Carpintería

Nuestro proyecto consiste en desarrollar una aplicación que permita gestionar presupuestos de productos de carpintería metálica. Para ello contamos con el siguiente esquema de datos:

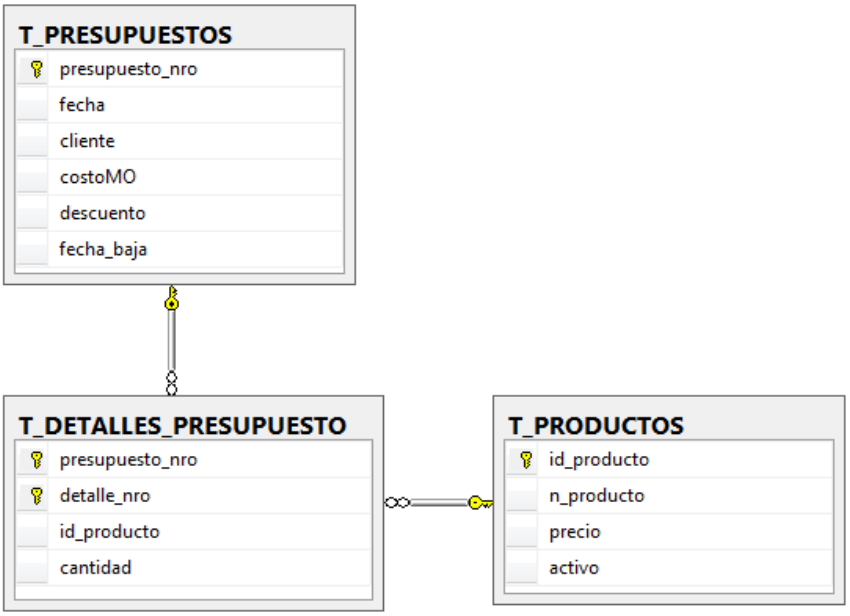

Imagen 4: Elaboración Propia

Si lo traducimos al modelo de Programación Orientada a Objetos (POO):

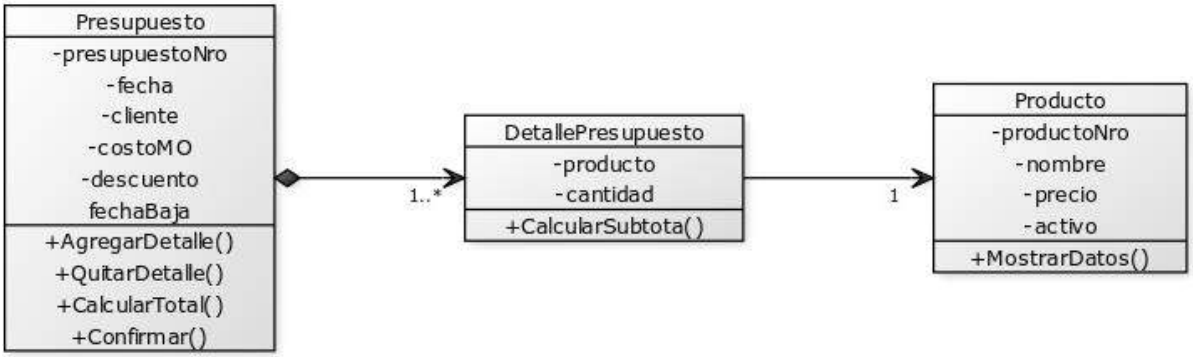

Imagen 5: Elaboración Propia

Creando la solución

Lo primero que vamos a crear es un proyecto en Visual Studio de tipo Aplicación de Windows Forms:

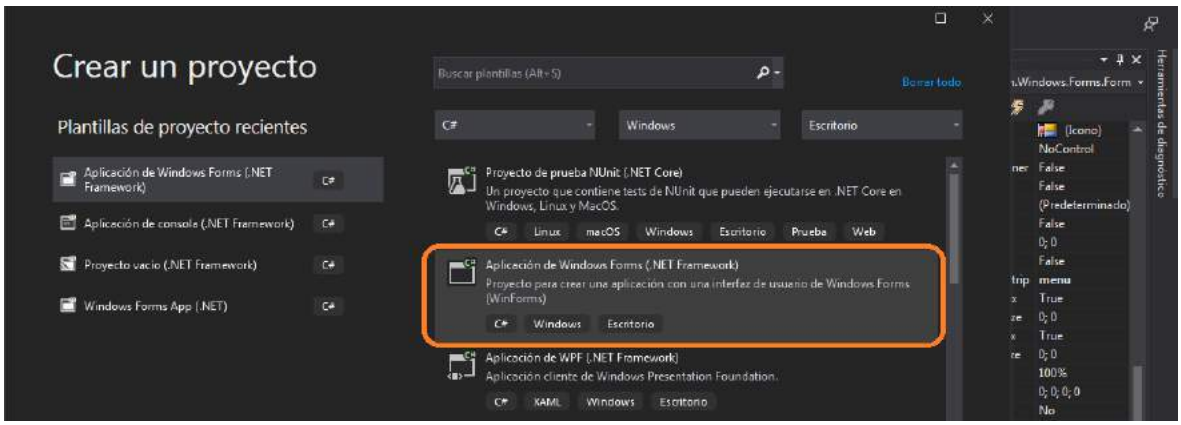

Imagen 6: Elaboración Propia

El formulario que nos crea por defecto lo dejaremos como formulario principal con un menú de opciones tal como se muestra en la siguiente imagen:

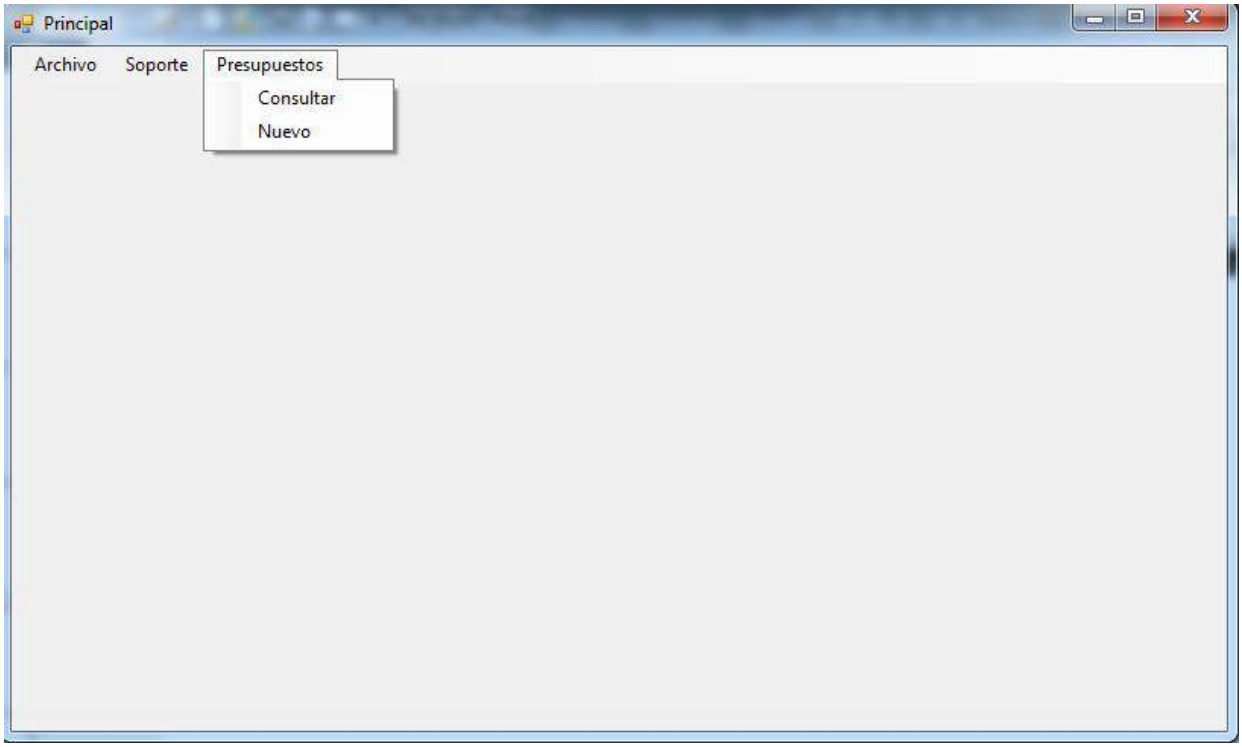

Imagen 7: Elaboración Propia

Este formulario debería tener un estado inicial maximizado (propiedad WindowState = Maximized). Por otra parte, el menú tiene las siguientes opciones:

- Archivo
  - Salir (Previa confirmación permite finalizar la aplicación)
- Soporte
  - Productos
    - Consultar (Permite consultar el listado completo de productos pre-cargados)
- Presupuesto
  - Consultar (Permite **consultar** los presupuestos cargados. Sobre un presupuesto sin datos de baja permite **modificar** un presupuesto o registrar los datos de **baja**)
  - Nuevo (Permite registrar el **alta** de un presupuesto)

Nuevo Presupuesto

Vamos a crear un formulario que permita crear un presupuesto (MAESTRO) con los productos y cantidades incluidos (DETALLE):

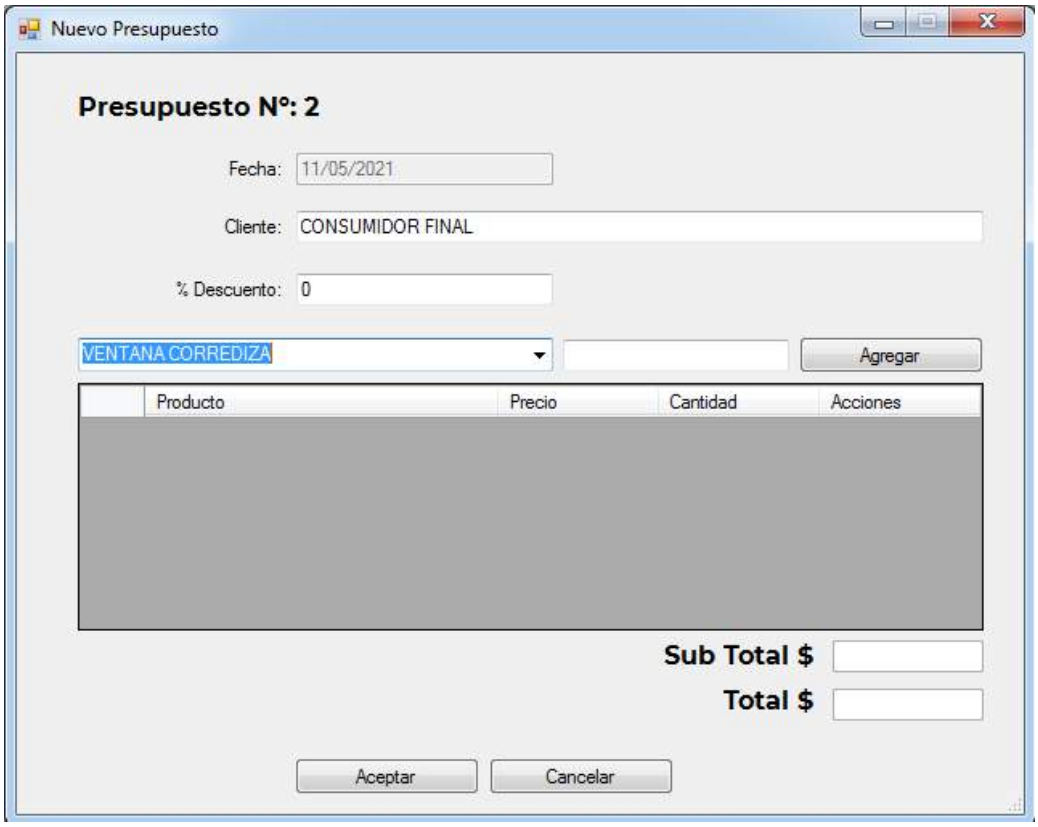

Imagen 8: Elaboración Propia

La pantalla permite seleccionar los productos a presupuestar desde una lista desplegable que se llena al cargar el formulario. A medida que se cargan los detalles del presupuesto se actualiza el subtotal y total con el descuento, si corresponde.

Eventos y código C#

- Lo primero que necesitamos es crear un objeto Presupuesto (entidad principal de nuestro modelo de negocio) cuando se inicializa el formulario:

```
public Frm_Alta_Presupuesto()
        {
            InitializeComponent();
            CargarProductos();
            //Crear nuevo presupuesto:
            presupuesto = new Presupuesto();
        }
```

- Al cargar el formulario:

```
private void Frm_Alta_Presupuesto_Load(object sender, EventArgs e)
        {
            ProximoPresupuesto();
            txtFecha.Text = DateTime.Now.ToString("dd/MM/yyyy");
            txtCliente.Text = "CONSUMIDOR FINAL";
            txtDto.Text = "0";
            this.ActiveControl = cboProductos; // Set foco al combo
        }
```

En este evento se inicializan los controles con los valores iniciales. Cabe resaltar el uso del método auxiliar `ProximoPresupuesto()`. Este método permite obtener el siguiente número de presupuesto mediante la llamada a un procedimiento llamado SP_PROXIMO_NRO_PRESUPUESTO, tal como se muestra a continuación:

```
private void ProximoPresupuesto()
        {
            try
            {
                SqlConnection cnn = new SqlConnection();
                cnn.ConnectionString = @"Data Source=.\SQLEXPRESS;Initial
                Catalog=carpinteria_db;Integrated Security=True";
                cnn.Open();
                SqlCommand cmd = new
                SqlCommand("SP_PROXIMO_NRO_PRESUPUESTO", cnn);
                cmd.CommandType = CommandType.StoredProcedure;
                SqlParameter param = new SqlParameter("@next",
                SqlDbType.Int);
                param.Direction = ParameterDirection.Output;
                cmd.Parameters.Add(param);
                cmd.ExecuteNonQuery();
                int next = Convert.ToInt32(param.Value);
                lblNroPresupuesto.Text = "Presupuesto Nº: " +
                next.ToString();
                cnn.Close();
            }
```

```
        catch (Exception)
        {
            MessageBox.Show("Error de datos", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
```

- Al seleccionar Agregar:

```
    private void btnAgregar_Click(object sender, EventArgs e)
    {
        if (cboProductos.Text.Equals(String.Empty))
        {
            MessageBox.Show("Debe seleccionar un PRODUCTO!",
            "Control", MessageBoxButtons.OK,
            MessageBoxIcon.Exclamation);
            return;
        }
        if (txtCantidad.Text == "" || !int.TryParse(txtCantidad.Text,
        out _))
        {
            MessageBox.Show("Debe ingresar una cantidad válida!",
            "Control", MessageBoxButtons.OK,
             MessageBoxIcon.Exclamation);
            return;
        }
        foreach (DataGridViewRow row in dgvDetalles.Rows)
        {
           if (row.Cells["colProd"].Value.ToString().Equals(cboProductos.Text))
           {
                MessageBox.Show("PRODUCTO: " + cboProductos.Text + "
                ya se encuentra como detalle!", "Control",
                MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
                return;
           }
        }
        DataRowView item = (DataRowView)cboProductos.SelectedItem;
```

```
            int prod = Convert.ToInt32(item.Row.ItemArray[0]);
            string nom = item.Row.ItemArray[1].ToString();
            double pre = Convert.ToDouble(item.Row.ItemArray[2]);
            Producto p = new Producto(prod, nom, pre);
            int cantidad = Convert.ToInt32(txtCantidad.Text);
            DetallePresupuesto detalle = new DetallePresupuesto(p,cantidad);
            presupuesto.AgregarDetalle(detalle);
            dgvDetalles.Rows.Add(new object[] { item.Row.ItemArray[0],
            item.Row.ItemArray[1], item.Row.ItemArray[2], txtCantidad.Text });
            CalcularTotal();
        }
```

En este evento básicamente se validan los datos de: **producto y cantidad**, y se controla que el mismo producto no se agregue más de una vez a la grilla. Notar que los datos completos del producto seleccionado en el combo se obtienen haciendo un casting del ítem seleccionado a `DataRowView`. Este objeto permite mediante la propiedad `Row` acceder a todos los datos recuperados por el procedimiento: SP_CONSULTAR_PRODUCTOS al cargar el combo de productos. Notar que la grilla guarda el dato **id_producto** en la columna 0 como no visible en tiempo de ejecución. Con estos datos se crea un objeto **Producto** que estará asociado a cada objeto **DetallePresupuesto** que se agrega como detalle al presupuesto creado al iniciar la pantalla.

Además, la grilla tiene un botón que se dibuja en la cuarta columna de cada fila. Este botón se declara en tiempo de diseño desde las propiedades del control `DataGridView`. Para gestionar el evento de dicho botón solo es necesario manejar un evento de click de la grilla y validar el índice de la celda correspondiente:

```
    private void dgvDetalles_CellContentClick(object sender, DataGridViewCellEventArgs
e)
        {
            if (dgvDetalles.CurrentCell.ColumnIndex == 4)
            {
                presupuesto.QuitarDetalle(dgvDetalles.CurrentRow.Index);
                //click button:
                dgvDetalles.Rows.Remove(dgvDetalles.CurrentRow);
                //presupuesto.quitarDetalle();
                CalcularTotal();
```

```
            }
        }
```

Por último se utiliza un método `CalcularTotal()` que actualiza los totales según los detalles ingresados.

```
private void CalcularTotal()
{
            double total = presupuesto.calcularTotal();
            txtTotal.Text = total.ToString();
            if(txtDto.Text != "")
            {
                double dto = (total * Convert.ToDouble(txtDto.Text)) / 100;
                txtFinal.Text = (total - dto).ToString();
            }
}
```

- Al seleccionar Cancelar:

```
        private void btnCancelar_Click(object sender, EventArgs e)
        {
            this.Dispose();
        }
```

Simplemente se cierra y se descarga la ventana de memoria mediante el método `Dispose()`, dejando activo nuevamente el formulario principal.

- Al seleccionar Aceptar:

```
        private void btnAceptar_Click(object sender, EventArgs e)
        {
            if (txtCliente.Text == "")
            {
                MessageBox.Show("Debe ingresar un cliente!", "Control",
                MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
                return;
            }
            if (dgvDetalles.Rows.Count == 0)
            {
                MessageBox.Show("Debe ingresar al menos detalle!",
                "Control", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
                return;
```

```
            }
            GuardarPresupuesto();
        }
```

Se valida que haya sido ingresado un cliente y que al menos la grilla tiene un detalle (para garantizar la relación MAESTRO-DETALLE) y se llama al método auxiliar **GuardarPresupuesto():**

```
private void GuardarPresupuesto()
        {
            presupuesto.Cliente = txtCliente.Text;
            presupuesto.Descuento = Convert.ToDouble(txtDto.Text);
            presupuesto.Fecha = Convert.ToDateTime(txtFecha.Text);
            if (presupuesto.Confirmar())
            {
                MessageBox.Show("Presupuesto registrado", "Informe",
                MessageBoxButtons.OK, MessageBoxIcon.Information);
                this.Dispose();
            }
            else {
                MessageBox.Show("ERROR. No se pudo registrar el
                presupuesto", "Error", MessageBoxButtons.OK,
                MessageBoxIcon.Error);
            }
        }
```

Lo más importante es el uso del método **Confirmar()** de la clase Presupuesto que es el responsable de materializar los datos en la base de datos, tal como se muestra a continuación:

```
public bool Confirmar() {
            bool resultado = true;
            SqlConnection cnn = new SqlConnection();
            try
            {
                cnn.ConnectionString = @"Data Source=.\SQLEXPRESS;Initial
                Catalog=carpinteria_db;Integrated Security=True";
                cnn.Open();
                SqlCommand cmd = new SqlCommand("SP_INSERTAR_MAESTRO",Cnn);
```

```
            cmd.CommandType = CommandType.StoredProcedure;
            cmd.Parameters.AddWithValue("@cliente", this.Cliente);
            cmd.Parameters.AddWithValue("@dto", this.Descuento);
            cmd.Parameters.AddWithValue("@total", this.calcularTotal() -
            this.Descuento);
            SqlParameter param = new SqlParameter("@presupuesto_nro",
            SqlDbType.Int);
            param.Direction = ParameterDirection.Output;
            cmd.Parameters.Add(param);
            cmd.ExecuteNonQuery();
            int presupuestoNro = Convert.ToInt32(param.Value);
            int cDetalles = 1;
            foreach (DetallePresupuesto det in Detalles)
            {
                SqlCommand cmdDet = new SqlCommand("SP_INSERTAR_DETALLE", cnn);
                cmdDet.CommandType = CommandType.StoredProcedure;
                cmdDet.Parameters.AddWithValue("@presupuesto_nro",
                presupuestoNro);
                cmdDet.Parameters.AddWithValue("@detalle", cDetalles);
                cmdDet.Parameters.AddWithValue("@id_producto",
                det.Producto.ProductoNro);
                cmdDet.Parameters.AddWithValue("@cantidad",
                det.Cantidad);
                cmdDet.ExecuteNonQuery();
                cDetalles++;
            }
        }
        catch (Exception ex)
        {
            resultado = false;
        }
        finally
        {
            if (cnn != null && cnn.State == ConnectionState.Open)
                cnn.Close();
```

```
            }
            return resultado;
        }
```

En este método se llaman a los procedimientos almacenados: SP_INSERTAR_MAESTRO y SP_INSERTAR_DETALLE para insertar el presupuesto (MAESTRO) y los detalles de presupuesto (DETALLES) respectivamente. Para ello se utilizan dos comandos diferentes. Notar que el primero insertar el maestro y devuelve el número del presupuesto insertado. Dicho valor será utilizado como clave en las inserciones de los detalles.

Si bien el código funciona correctamente sería interesante plantearse qué sucedería si el primer comando se ejecuta exitosamente, pero los segundos fallan. La base de datos quedaría en un estado inconsistente ya que existiría una fila en la tabla T_PRESUPUESTO, pero no tendría ninguna fila de T_DETALLES_PRESUPUESTO que se correspondiera con ella (es decir un MAESTRO sin DETALLES). Esta situación podría fácilmente ser resuelta si incorporamos el concepto de **transacción.**

MANEJO DE TRANSACCIONES

Las transacciones se usan cuando se desea realizar una secuencia de operaciones (o sentencias SQL), de manera que son ejecutadas como una única **unidad de trabajo**. Por ejemplo, queremos una aplicación que realice dos acciones en nuestra base de datos, la primera acción inserta un registro en nuestra tabla MAESTRO y la otra acción inserta un segundo registro en la tabla de DETALLE. Con el manejo de transacciones de ADO.NET, no importa que cualquiera de las dos acciones falle, ya que los cambios no se realizarán en la base de datos a menos que todas las operaciones sean **confirmadas de manera exitosa.**

Las propiedades de las transacciones se las conoce como **ACID**: **A**tomicidad, **C**oherencia, A**i**slamiento y **D**urabilidad.

- *Atomicidad*: Una transacción debe ser una unidad atómica de trabajo, o se hace todo o no se hace nada.
- *Coherencia*: Debe dejar los datos en un estado coherente luego de realizada la transacción.
- *Aislamiento*: Las modificaciones realizadas por transacciones son tratadas en forma independiente, como si fueran un solo y único usuario de la base de datos.

- *Durabilidad*: Una vez concluida la transacción sus efectos son permanentes y no hay forma de deshacerlos.

Refactorizando el método Confirmar()

ADO.NET soporta el concepto de transacción mediante el uso transacciones iniciadas por código. Para llevar a cabo una transacción de este tipo son necesarios los siguientes pasos:

1. Llamar al método **BeginTransaction** del objeto **SqlConnection** para marcar el comienzo de la transacción. El método BeginTransaction devuelve una referencia a la transacción. Esta referencia se asigna a los objetos **SqlCommand** que están inscritos en la transacción.
2. Asigne el objeto Transaction a la propiedad Transaction del objeto SqlCommand que se va a ejecutar. Si el comando se ejecuta en una conexión con una transacción activa y el objeto Transaction no se ha asignado a la propiedad Transaction del objeto Command, se inicia una excepción.
3. Ejecute los comandos necesarios.
4. Llame al método Commit del objeto SqlTransaction para completar la transacción, o al método Rollback para finalizarla. Si la conexión se cierra o elimina antes de que se hayan ejecutado los métodos Commit o Rollback, la transacción se revierte.

Si utilizamos este criterio en el método **Confirmar()** el código resulta:

```
public bool Confirmar() {
     bool resultado = true;
     SqlConnection cnn = new SqlConnection();
     SqlTransaction t = null;
     try
     {
         cnn.ConnectionString = @"Data Source=.\SQLEXPRESS;Initial
         Catalog=carpinteria_db;Integrated Security=True";
         cnn.Open();
         t = cnn.BeginTransaction();
         SqlCommand cmd = new SqlCommand("SP_INSERTAR_MAESTRO", cnn, t);
         cmd.CommandType = CommandType.StoredProcedure;
         cmd.Parameters.AddWithValue("@cliente", this.Cliente);
         cmd.Parameters.AddWithValue("@dto", this.Descuento);
```

```
            cmd.Parameters.AddWithValue("@total", this.calcularTotal() -
            this.Descuento);
            SqlParameter param = new SqlParameter("@presupuesto_nro",
            SqlDbType.Int);
            param.Direction = ParameterDirection.Output;
            cmd.Parameters.Add(param);
            cmd.ExecuteNonQuery();
            int presupuestoNro = Convert.ToInt32(param.Value);
            int cDetalles = 1;
            foreach (DetallePresupuesto det in Detalles)
            {
                SqlCommand cmdDet = new
                SqlCommand("SP_INSERTAR_DETALLE", cnn, t);
                cmdDet.CommandType = CommandType.StoredProcedure;
                cmdDet.Parameters.AddWithValue("@presupuesto_nro",
                presupuestoNro);
                cmdDet.Parameters.AddWithValue("@detalle", cDetalles);
                cmdDet.Parameters.AddWithValue("@id_producto",
                det.Producto.ProductoNro);
                cmdDet.Parameters.AddWithValue("@cantidad",
                det.Cantidad);
                cmdDet.ExecuteNonQuery();
                cDetalles++;
            }
            t.Commit();
        }
        catch (Exception ex)
        {
            t.Rollback();
            resultado = false;
        }
        finally
        {
            if (cnn != null && cnn.State == ConnectionState.Open)
                cnn.Close();
```

```
            }
            return resultado;
        }
```

Solo resta completar las funcionalidades de: consulta, modificación y baja de presupuestos. Dicha implementación quedará como material anexo a esta unidad, aunque es altamente recomendable usarlo como actividad de auto-aprendizaje por parte de los estudiantes.

REPORTES EN C#

En el ámbito de la informática, los reportes son salidas de un sistema que organizan y exhiben 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.

El reporte, de esta forma, confiere una mayor utilidad a los datos. No es lo mismo trabajar con una planilla de cálculos con 10.000 campos que con un dibujo en forma de torta que presenta dichos campos de manera gráfica. De la misma forma, gracias a los reportes cualquier persona puede proceder a realizar un resumen de datos o a clasificar estos en grupos determinados. Así, teniendo en cuenta los datos que abordan y la extensión que tienen, estos reportes pueden clasificarse en:

- **Listados**: generalmente utilizados para dar soporte a procesos operativos utilizados por los niveles más bajos dentro de la estructura organizacional. Son ejemplos de este tipo de reportes: los listados de pedidos pendientes a la fecha, una hoja de ruta, un listado de control de existencias.
- **Reportes:** utilizados para dar soporte a las necesidades de información de mandos medios permiten, como se indicó anteriormente, clasificar los datos en grupos (denominados en los lenguajes estructurados como cortes de control) y consolidar información de cantidades y/o importes. Éstos incluyen generalmente filtros por fechas, áreas, estados o tipos. Son ejemplos de reportes: reportes de ventas por sucursal y zona, reportes de vuelos por destino o reportes de mantenimiento por tipo de vehículo por mes.
- **Estadísticos:** son salidas que se utilizan fundamentalmente para dar soporte a las decisiones estratégicas mediante la consolidación de grandes volúmenes de datos transaccionales, utilizando gráficos (de barras o de torta) para representar los datos consultados. Por lo general se utilizan para

analizar la evolución de indicadores de gestión, comparativos de comportamientos de variables de interés entre períodos (anuales) o tendencias de variables críticas de negocio. Son ejemplos de estadísticas: Reporte interanual de ventas o un reporte anual de ganancias netas por sucursal.

Pasos para Generar un Reporte

En general para generar nuestros reportes podemos convenir los siguientes pasos:

1. Como primer paso crear un objeto DATASET [1] que utilizaremos como fuente de datos para diseñar el reporte y posteriormente para poblar con datos. Para crear un DATASET hacer click derecho sobre la carpeta Reports (dentro de GUILayer) y seleccionar *agregar>>nuevo elemento…*

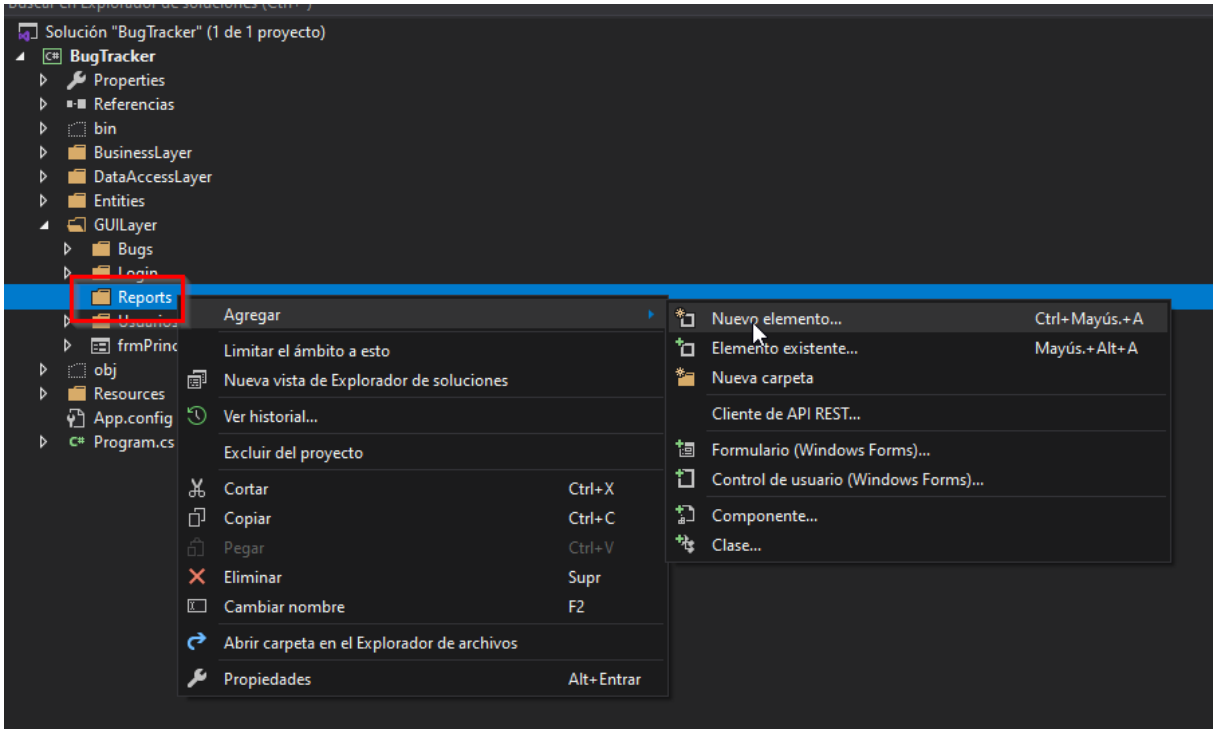

Imagen 9: Elaboración Propia

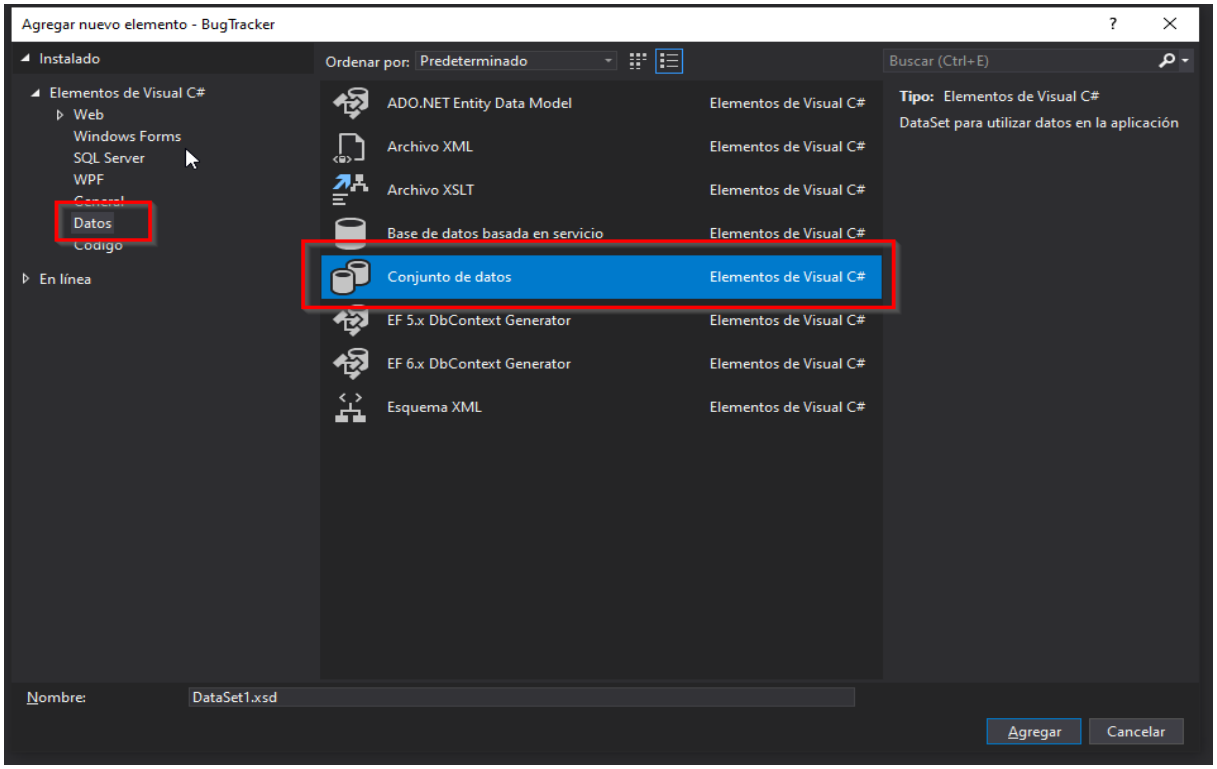

Imagen 10: Elaboración Propia

Una vez creado el DATASET haciendo click derecho sobre la pantalla de visualización del conjunto de datos nuevamente seleccionamos *agregar*, tal como se indica en la siguiente figura:

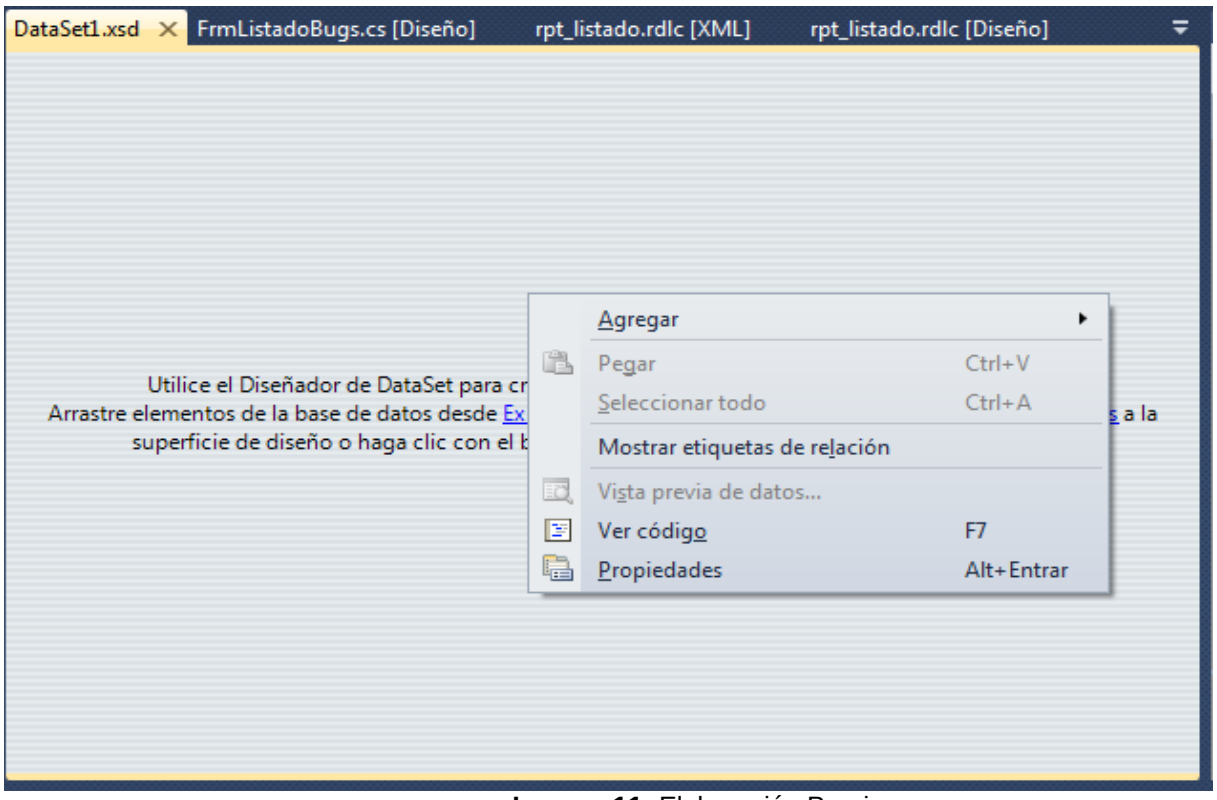

Imagen 11: Elaboración Propia

En este punto tenemos básicamente dos opciones:

- **TableAdapter**. Con esta opción podemos generar nuestra tabla y mediante el generador de consultas componer a partir de dos o más tablas de nuestra base de datos. Esta opción será utilizada en la sección siguiente: LISTADOS UTILIZANDO ASISTENTE PARA INFORMES.
- **Tabla de datos**. Con esta opción creamos un DATATABLE propio, al que necesitaremos definir columnas y tipos de datos (como si fuese una nueva tabla). A diferencia de lo anterior, la consulta SELECT con la que llenaremos este objeto será definida por fuera del DATASET y dinámicamente, en tiempo de ejecución, será asignada por código antes de visualizar el reporte. Esta opción será utilizada en la sección REPORTES.

2. El segundo paso será diseñar nuestro reporte. Para ello hacemos click derecho sobre la carpeta Reports, *agregar>>nuevo elemento…*

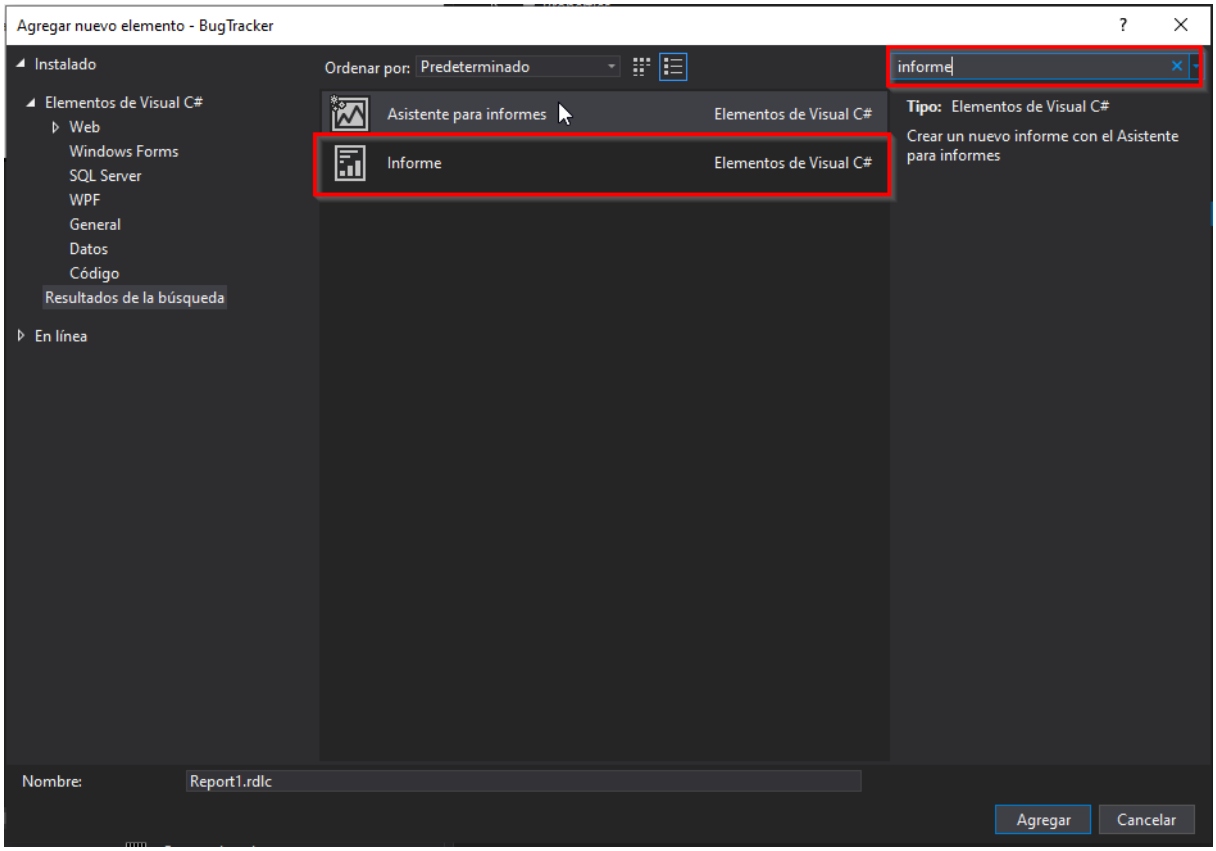

Imagen 12: Elaboración Propia

Nota: Si no está disponible este elemento, consultar en anexo: Instalación de Microsoft RDLC Report Designer.

Se nos crea un área de trabajo donde podremos arrastrar y soltar controles: como tablas, cuadros de textos, líneas o imágenes, de la misma forma que si diseñamos un formulario *WinForm.*

Se deberá habilitar la opción *Ver>>Cuadro de herramientas,* en caso de que no se muestre por defecto, para disponer de la paleta de controles.

La siguiente figura muestra el área de diseño de nuestro reporte:

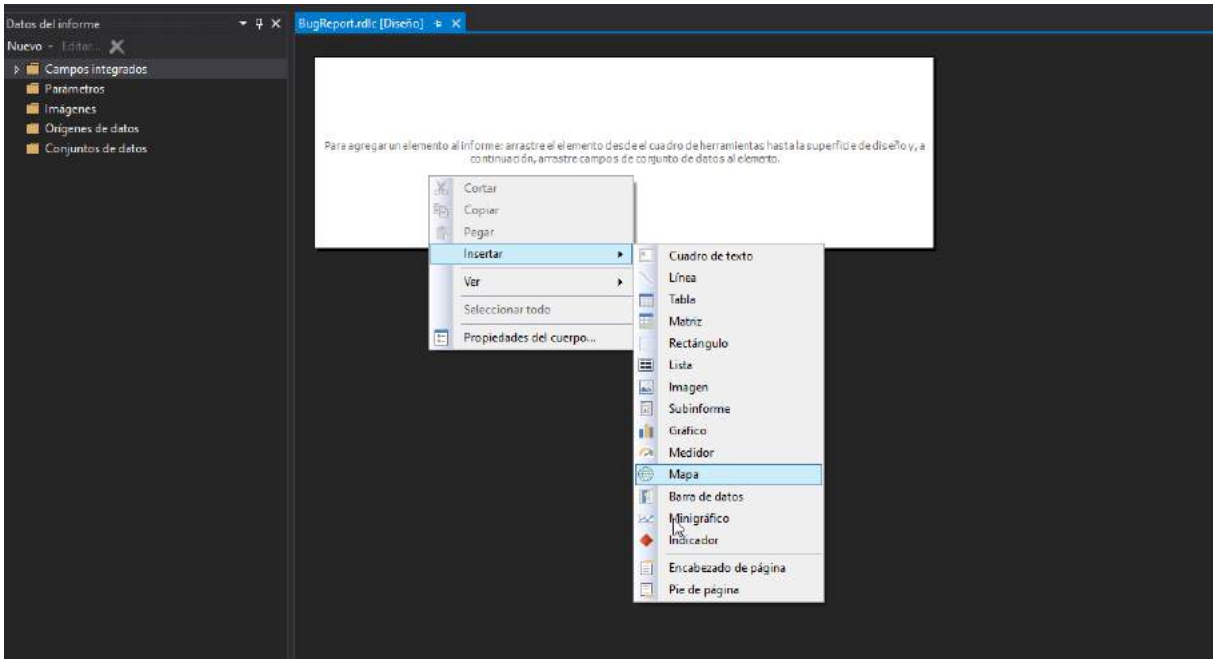

Imagen 13: Elaboración Propia

Por defecto un reporte tiene solo un cuerpo, pero podemos agregar un Encabezado y un Pie de página:

- **Encabezado**: esta sección se utiliza para mostrar datos que no se repiten, tales como el título del reporte, la fecha de generación o el logo de la empresa. Preferentemente los datos de esta sección solo deberían mostrarse en la primera página del reporte. Para indicar esto último hacemos click derecho sobre el encabezado y mediante la opción *propiedades del encabezado…* podemos fijar este comportamiento.
- **Cuerpo**: permite incluir los datos que se repiten, es decir, debería contener los datos obtenidos del SELECT a la base de datos. Acá se incluyen generalmente las tablas de resultados. Cuando arrastramos una tabla automáticamente el IDE nos permite seleccionar un conjunto de datos para elegir nuestra tabla

- **Pié de página**: similar a la sección de encabezado pero al pié del informe. Generalmente incluimos el número de página o el nombre del reporte. Esta sección debería repetirse en cada página del documento.

Algo interesante para resaltar es la posibilidad de insertar en un control (generalmente un cuadro de texto o en una columna de tabla) la opción Expresión. Mediante esta opción es posible acceder a un abanico de fórmulas y funciones integradas con las que podemos hacer un diseño robusto, sumamente flexible y fácil de mantener. Queda a cargo del alumno investigar y utilizar todo el potencial de esta herramienta. Si alguna vez utilizó el asistente de fórmulas de una planilla de cálculos EXCEL, entonces se sentirá familiarizado con esta opción.

3. Por último necesitamos visualizar nuestro diseño (objeto.rdlc) sobre un control específico para mostrar reportes. Vamos a crear un formulario (WinForm) y agregar un control de reporte desde el cuadro de herramientas: **ReportViewer**.

Nota: Si no está disponible este elemento, consultar en anexo: Instalación de paquete NuGet para el control ReportViewer.

4. Cuando se arrastra este componente, Visual Studio automáticamente agrega en el evento Load() del formulario:

```
this.reportViewer1.RefreshReport();
```

Está línea permite ejecutar el reporte y visualizar el diseño realizado en nuestro objeto .rdlc. Para que esto último funcione correctamente debemos pararnos sobre el objeto reportViewer y enlazarlo con nuestro diseño. Tal como se muestra en la siguiente figura:

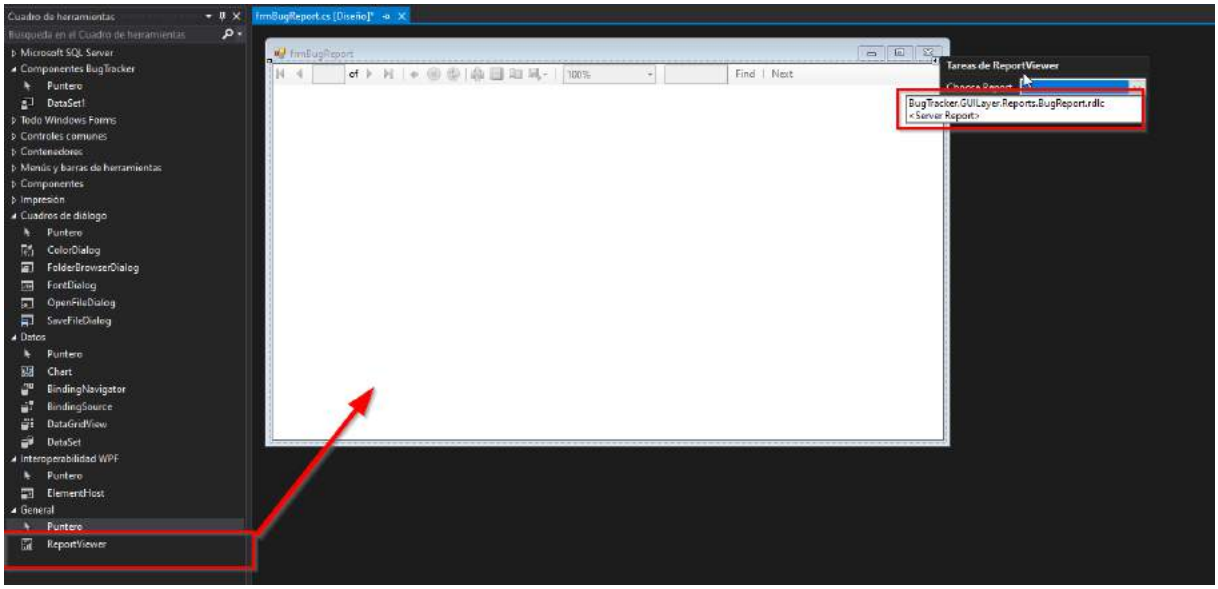

Imagen 14: Elaboración Propia

Listados utilizando un TableAdapter

Para crear un listado simple vamos a utilizar los siguientes pasos:

1. Primero vamos a crear un DATASET llamado *DSListado.xsd* y luego vamos a hacer click derecho y seleccionar la opción *agregar>>tableAdapter.*

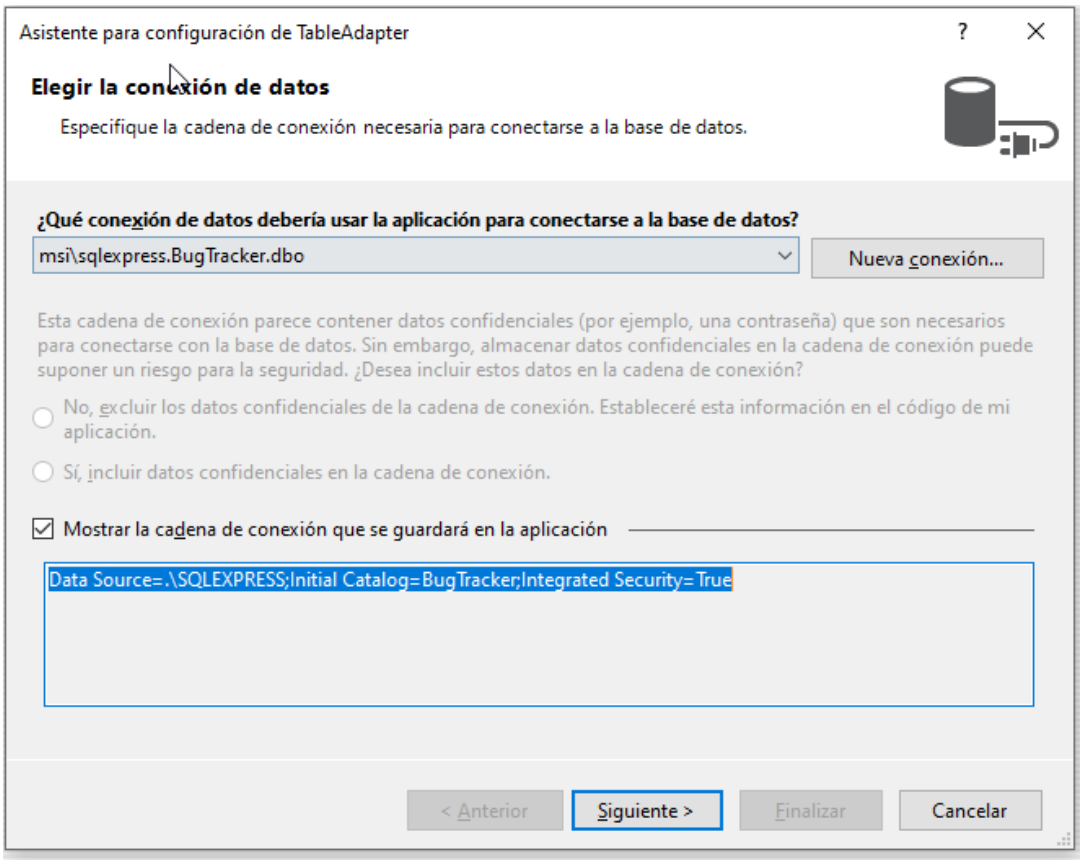

Imagen 15: Elaboración Propia

- Lo primero que nos pide es una cadena de conexión a nuestra base. Si no la hemos generado aún podemos seleccionar la opción Nueva conexión… y seguir los pasos del asistente.
- Definida la conexión seleccionamos Siguiente:

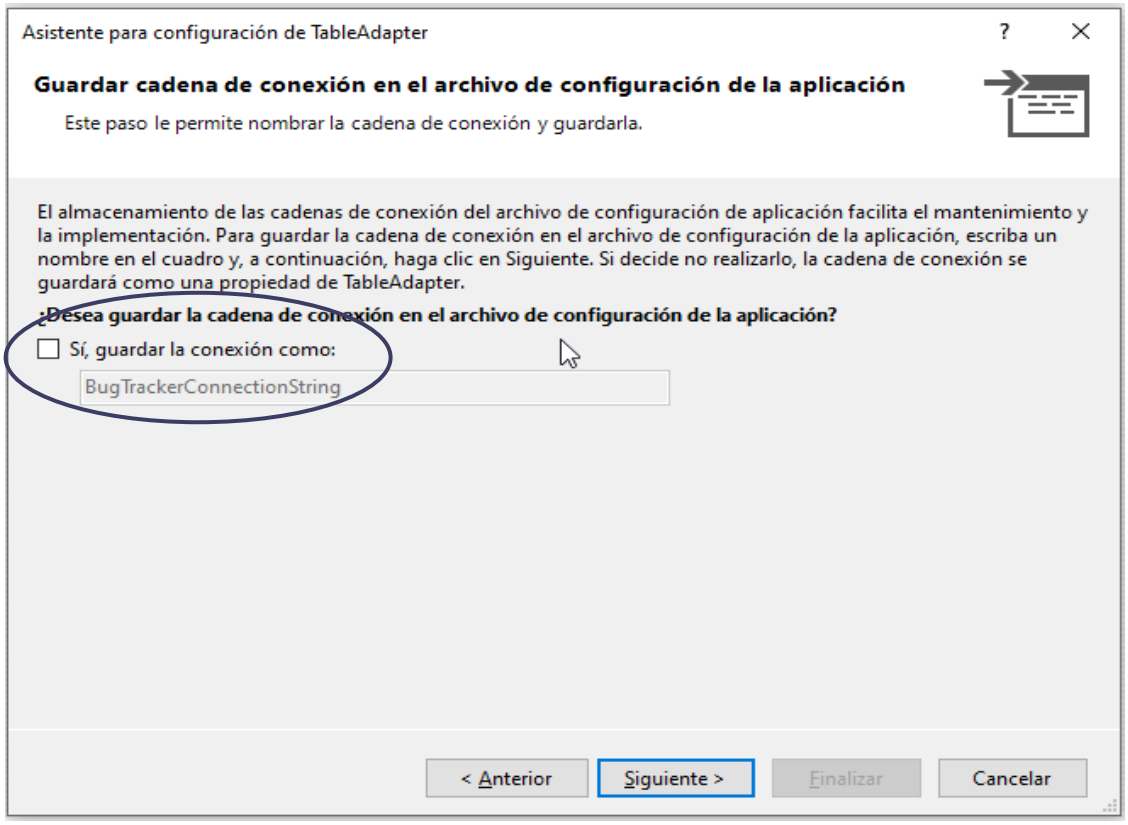

Imagen 16: Elaboración Propia

- Destildamos la opción “guardar la conexión como“ seleccionamos Siguiente:

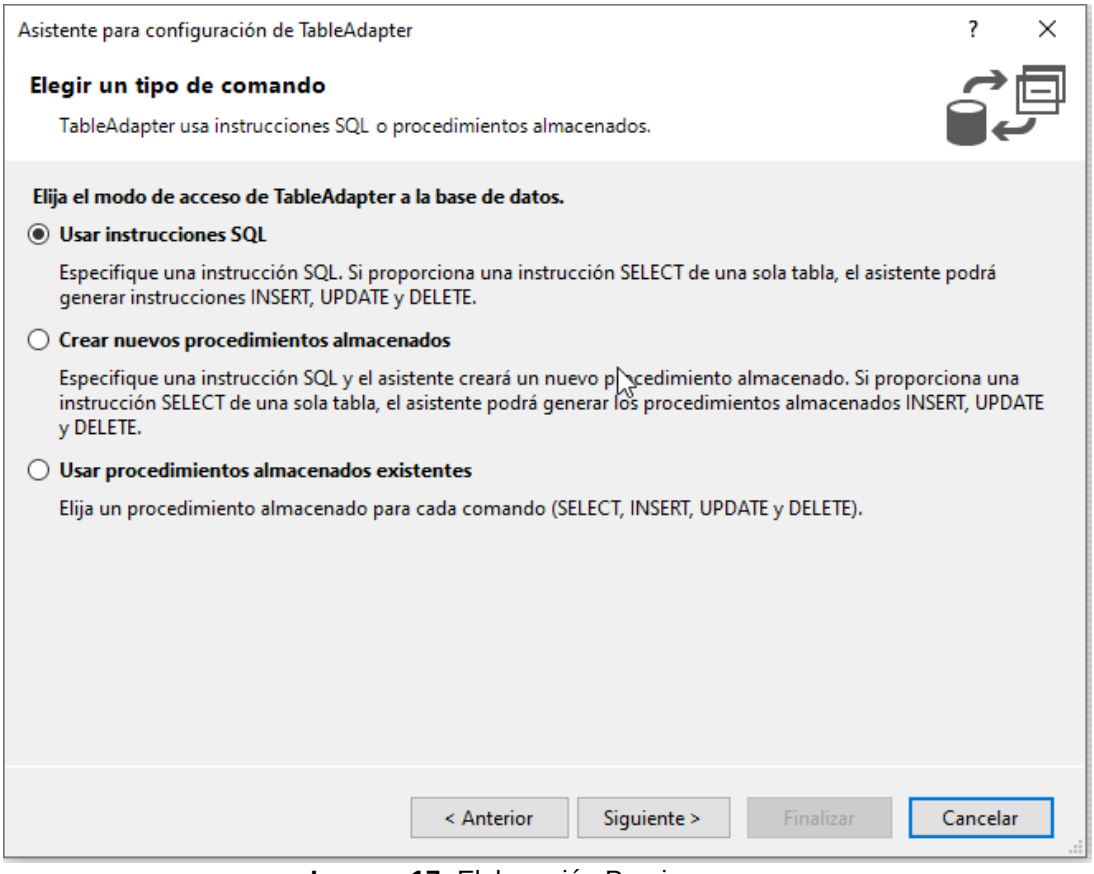

Imagen 17: Elaboración Propia

- Seleccionamos la primera opción: *Usar instrucciones SQL.* Notar que también podemos utilizar un procedimiento almacenado existente en nuestra base de datos o crear uno desde el asistente. En la siguiente pantalla seleccionamos la opción *Generador de consultas…*

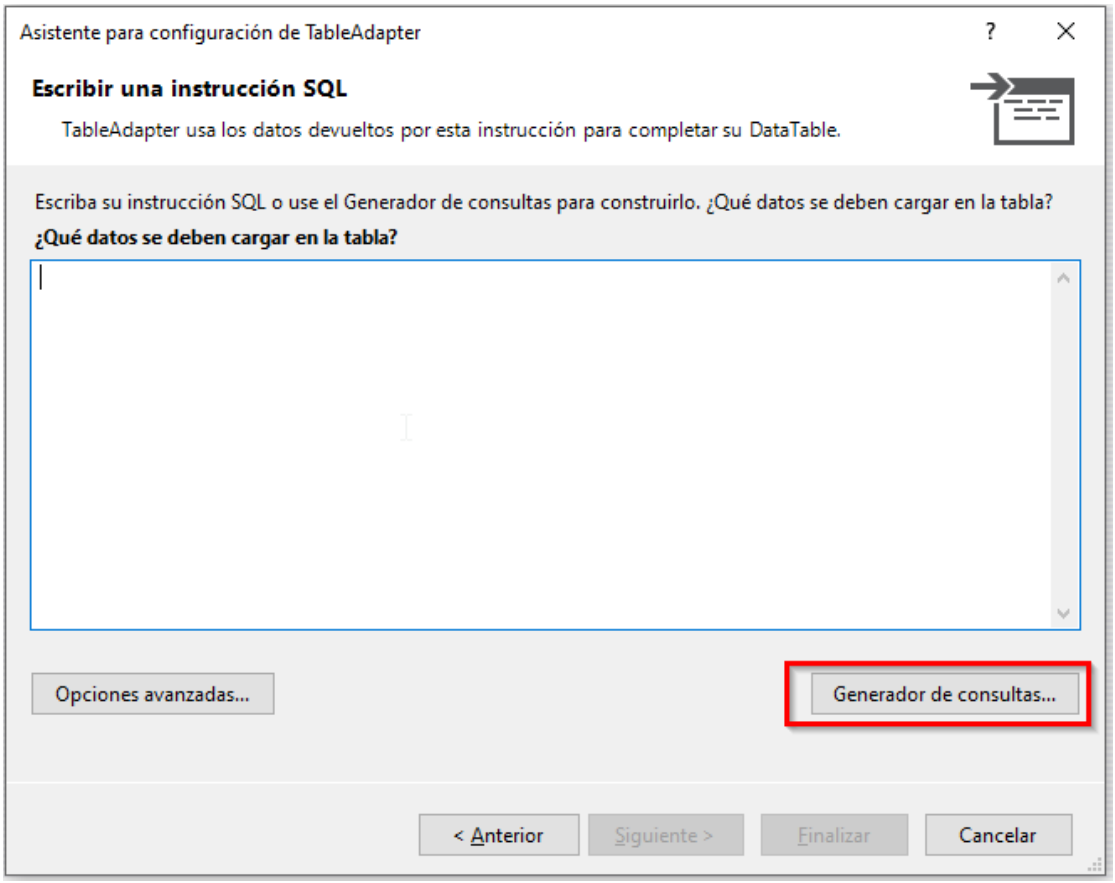

Imagen 18: Elaboración Propia

- Esta opción nos permite crear una tabla en memoria que surja de la conjunción de varias tablas de nuestra base. Al seleccionar las tablas el asistente automáticamente arma la sentencia SQL con los INNER JOIN respectivo respetando las claves primarias y foráneas definidas en nuestro modelo de datos.

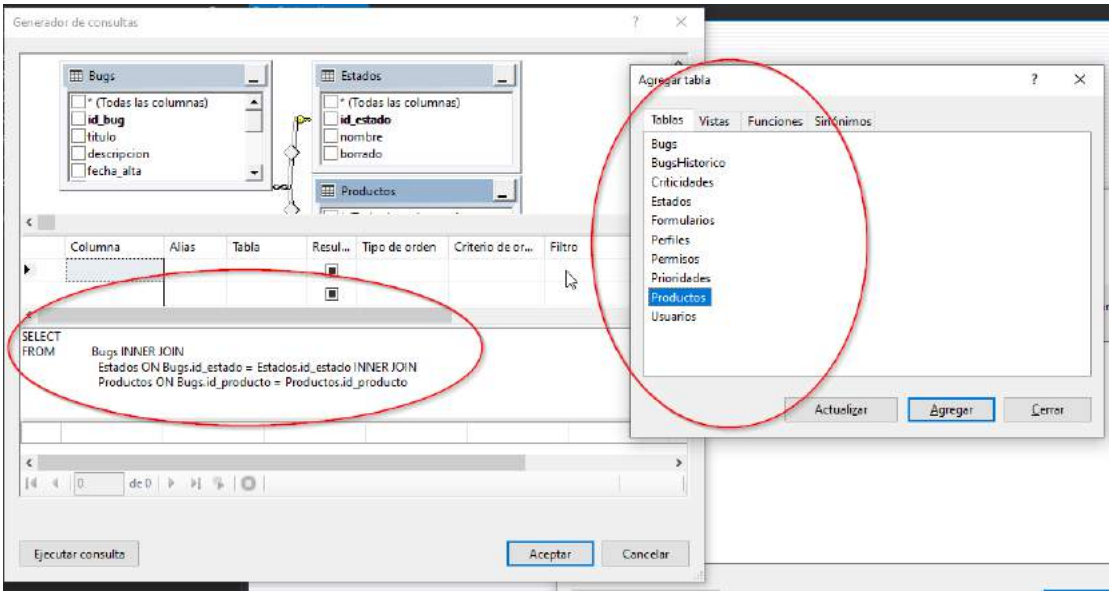

Imagen 19: Elaboración Propia

- Luego seleccionamos las columnas de las tablas a incluir en el SELECT y con la opción Ejecutar consulta podemos validar los resultados obtenidos.
- La siguiente pantalla permite configurar los métodos que tendrá el objeto TableAdapter para poder llenar el DATATABLE con la consulta generada.

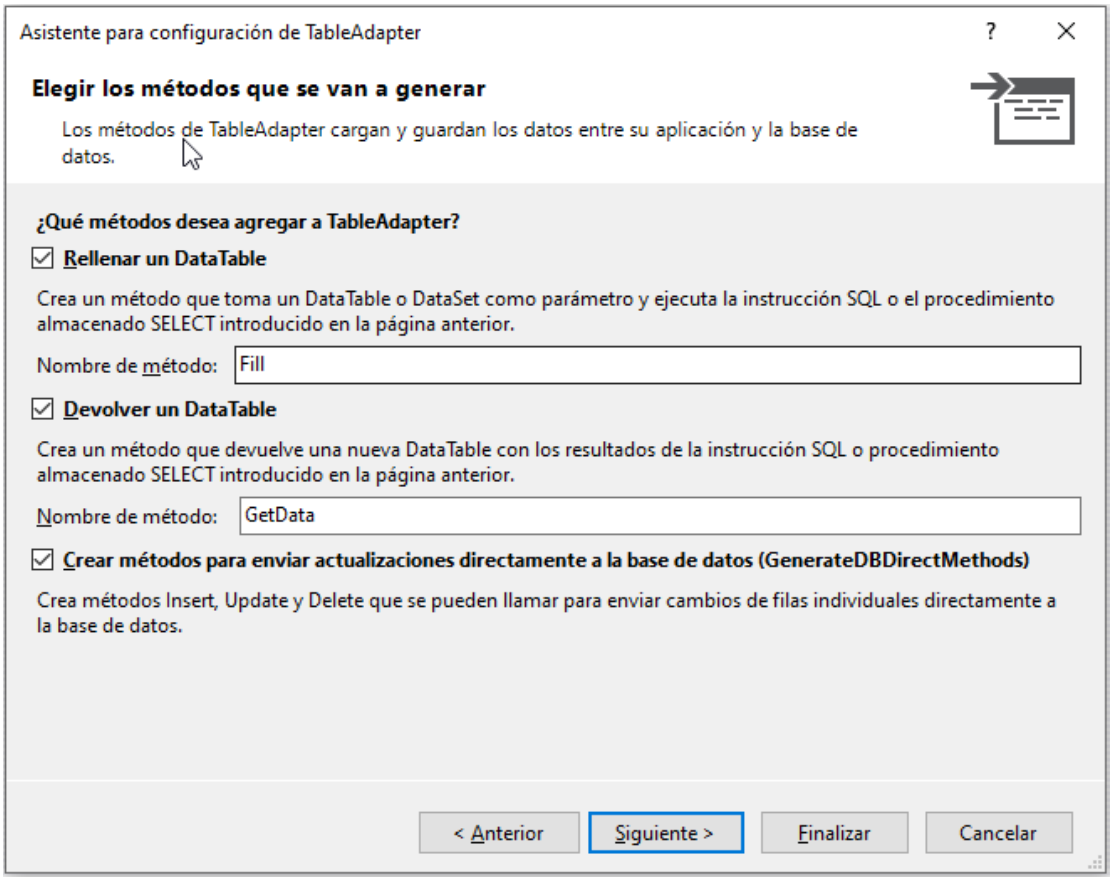

Imagen 20: Elaboración Propia

- Al finalizar el asistente tendremos ya configurado nuestro DATASET para usarlo en el siguiente paso.

2. El siguiente paso es definir nuestro archivo .rdlc con el diseño del reporte. Para ellos seleccionamos *agregar>>nuevo elemento*, y en la categoría Reporting seleccionamos **Informe.** Luego hacemos click derecho opción *insertar* y seleccionamos **Encabezado y Pié de página.**

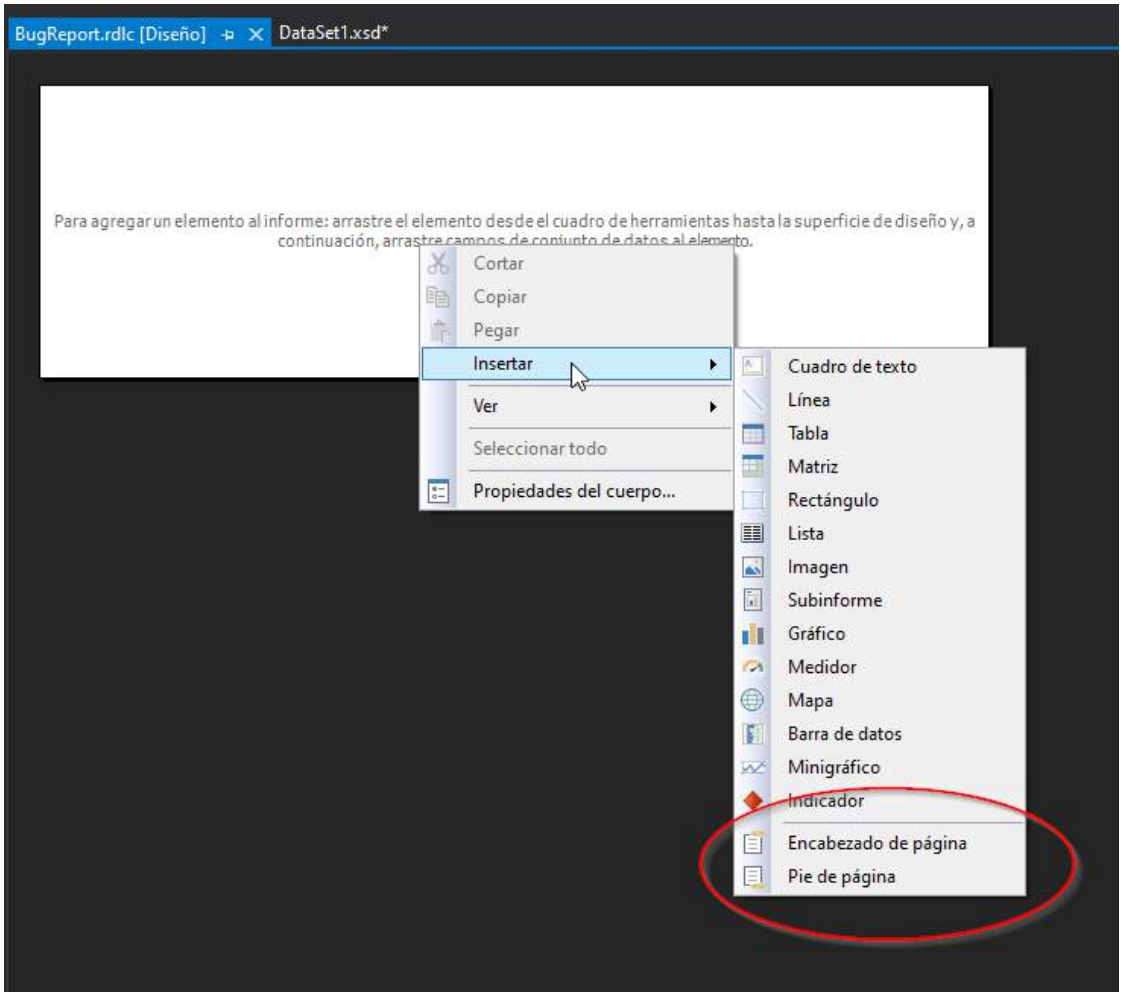

Imagen 21: Elaboración Propia

- Luego de agregar las secciones mínimas del reporte, arrastramos y soltamos desde el cuadro de herramientas una tabla a la sección central.

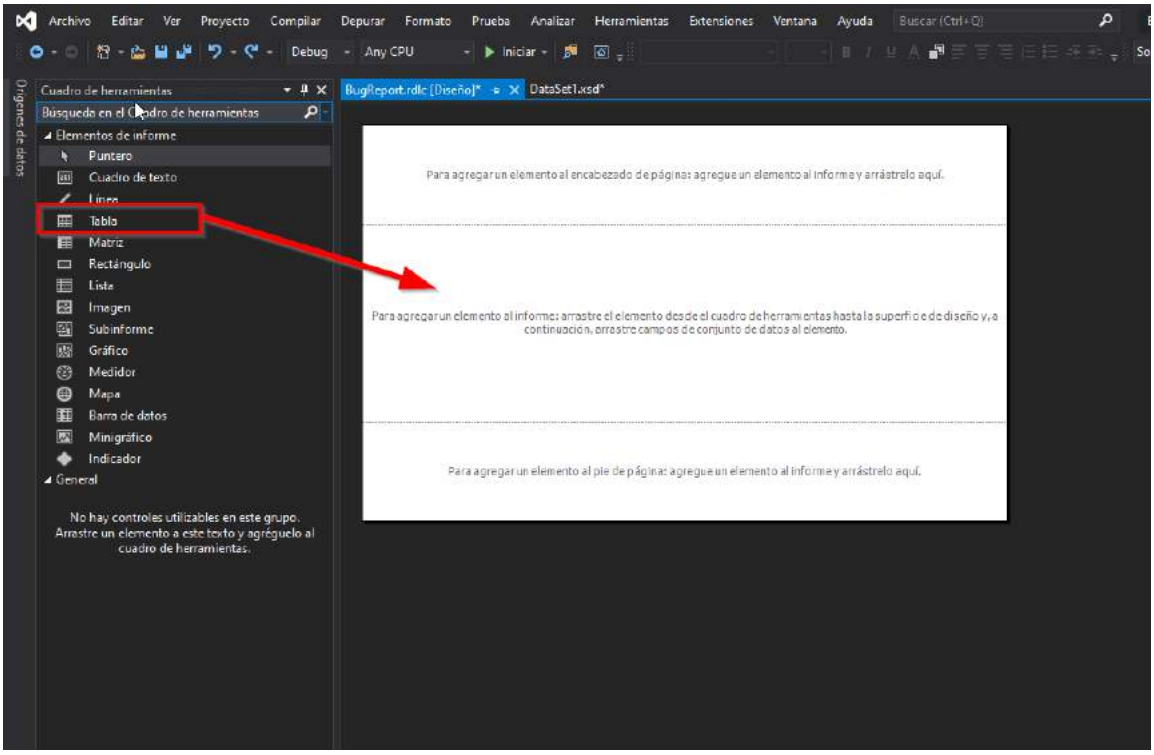

Imagen 22: Elaboración Propia

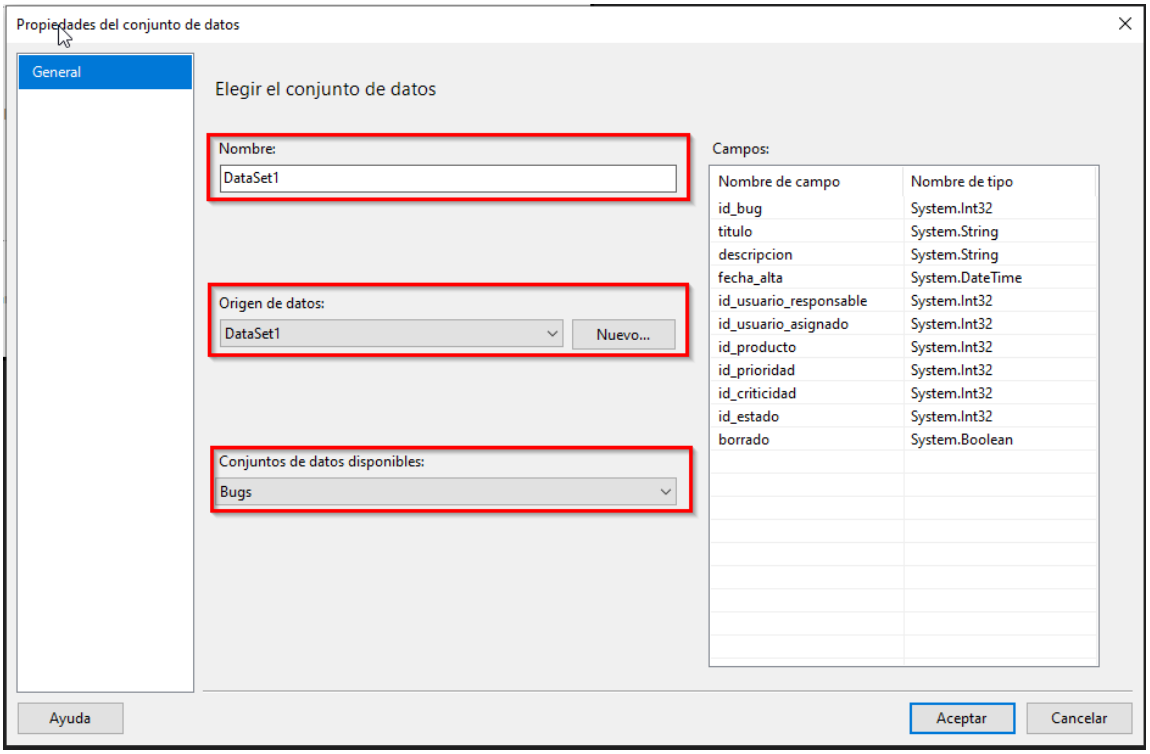

Imagen 23: Elaboración Propia

- Esta pantalla nos permite asociar el DATASET creado en el paso anterior con nuestro diseño. Como se observa podemos elegir DataSet1 y seleccionar el objeto Table generado por el TableAdapter..

- **Importante**: El nombre que se le asigne a este conjunto de datos será el que luego podrá referenciarse por código en tiempo de ejecución.
- Con esto ya podemos armar el detalle de los datos del reporte en función de las columnas del datatable. Como se mencionó anteriormente se sugiere investigar sobre las *opciones de Expresión* que vienen con el VS para agregar elementos al reporte. Si por ejemplo necesitáramos el total de filas recuperadas podemos arrastrar un Cuadro de texto y sobre el componente hacemos click derecho opción *Expresión:*

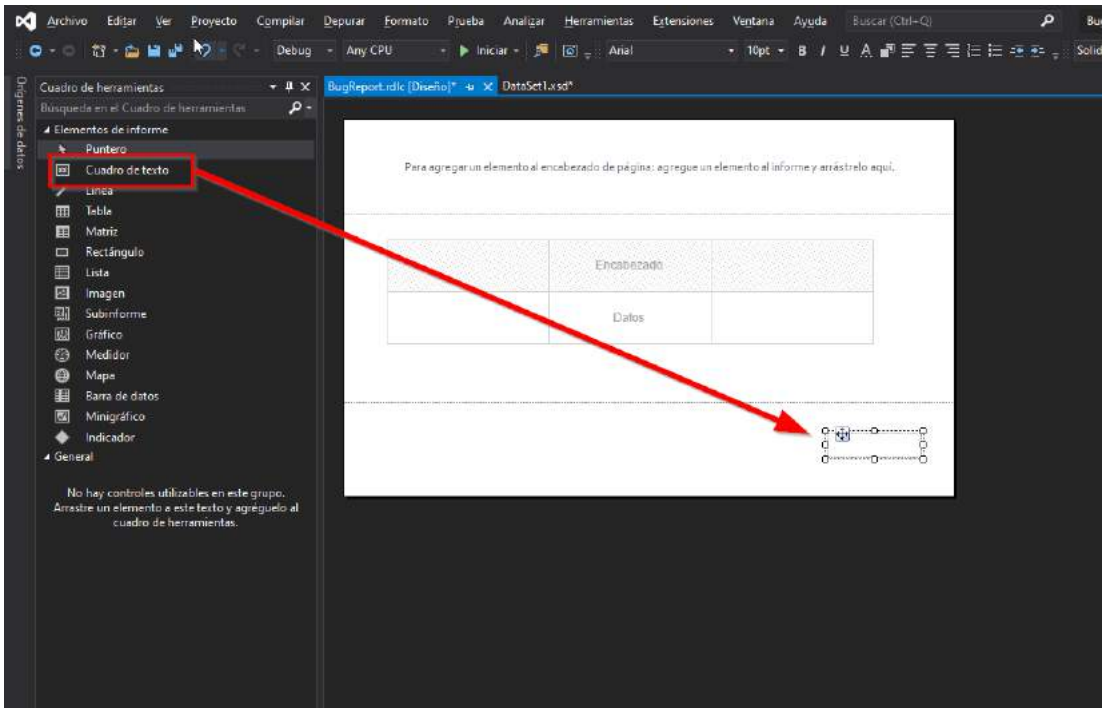

Imagen 24: Elaboración Propia

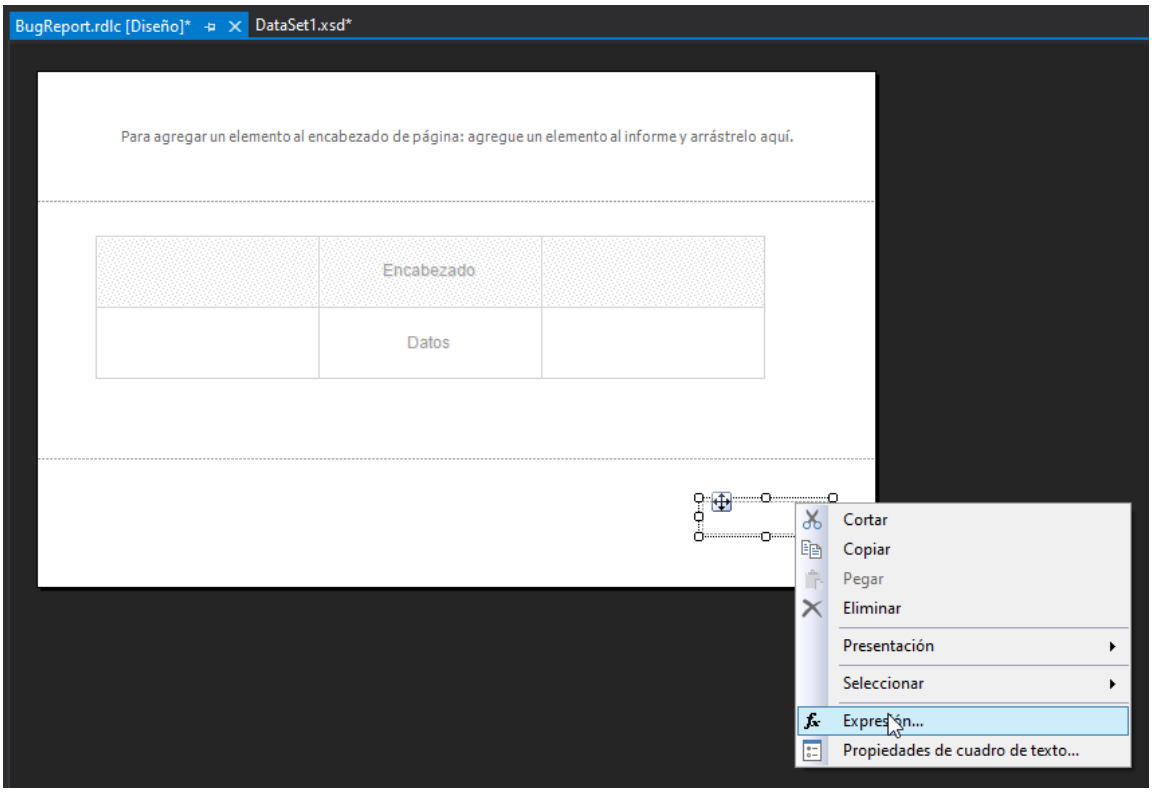

Imagen 25: Elaboración Propia

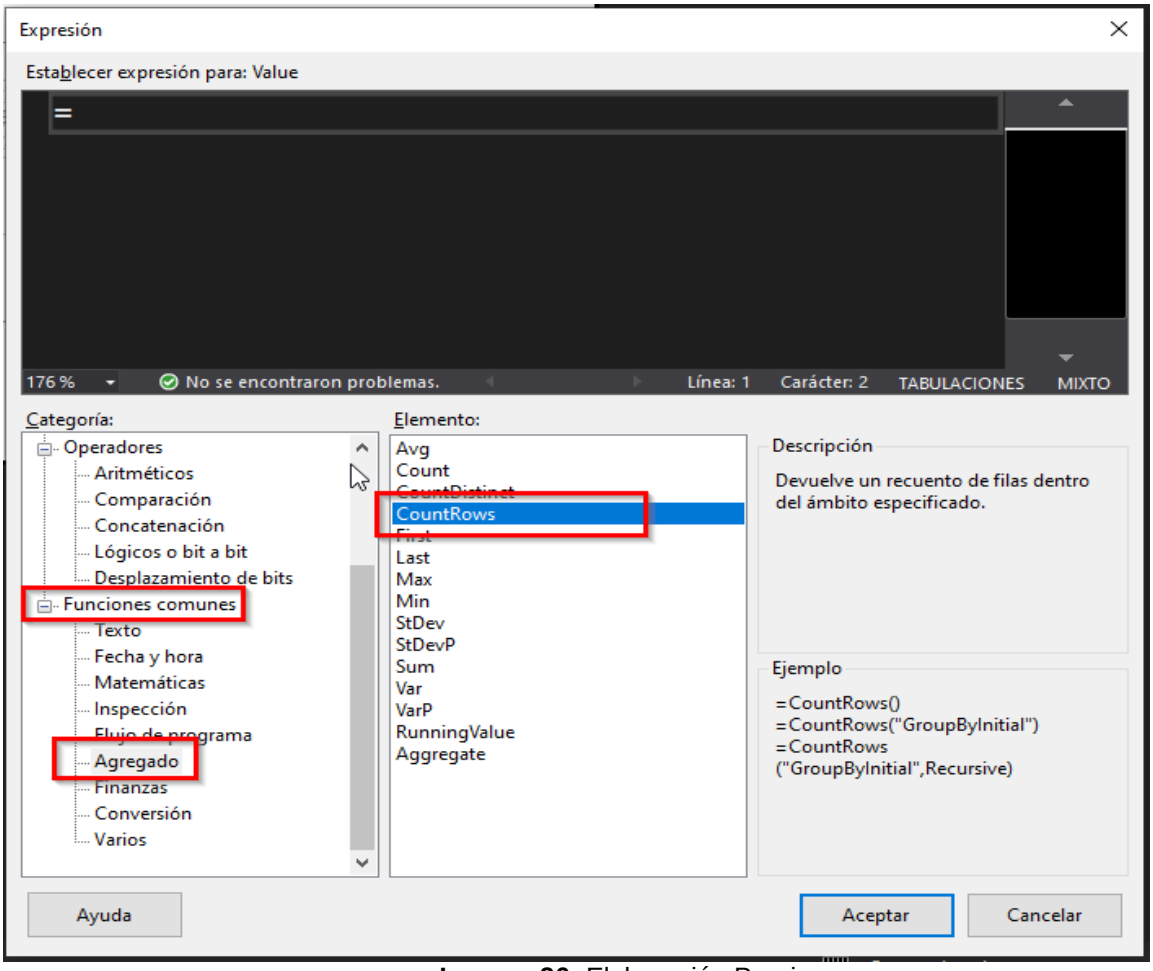

Imagen 26: Elaboración Propia

- En la categoría *Funciones comunes* seleccionamos CountRows() y podemos generar una expresión que cuente la cantidad de filas del reporte mediante esta función.

3. Por último vamos a crear un formulario donde incluiremos un control **ReportViewer.**

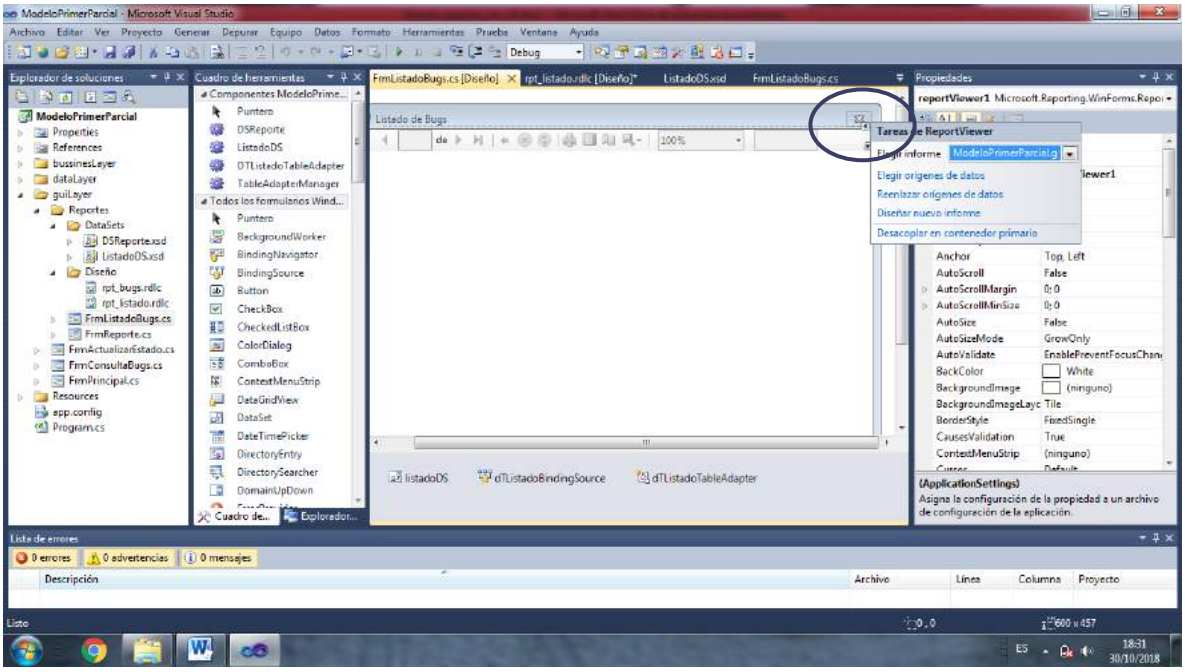

Imagen 27: Elaboración Propia

Luego hacemos click en el botón indicado en la imagen y seleccionamos el objeto .rdlc que será asociado con este visualizador de reportes.

En el evento Load() del formulario se agrega automáticamente la siguiente línea de código:

```
this.BugsTableAdapter.Fill(this.DataSet1.Bugs);
```

Esta línea permite poblar el dataset del reporte con el datatable creado por el tableadapter del paso 1.

Es importante resaltar que no hemos escrito ni una sola línea de código y el reporte queda funcionando. En la sección siguiente veremos cómo hacer esto mismo escribiendo código y asignando los objetos en tiempo de ejecución.

Reportes utilizando una Tabla propia

Para crear un reporte personalizado con uno o más grupos o cortes de control vamos a utilizar los siguientes pasos:

1. Primero vamos a crear un DATASET llamado *DSReporte.xsd* y luego vamos a hacer click derecho y seleccionar la opción *agregar>>Tabla de datos.*

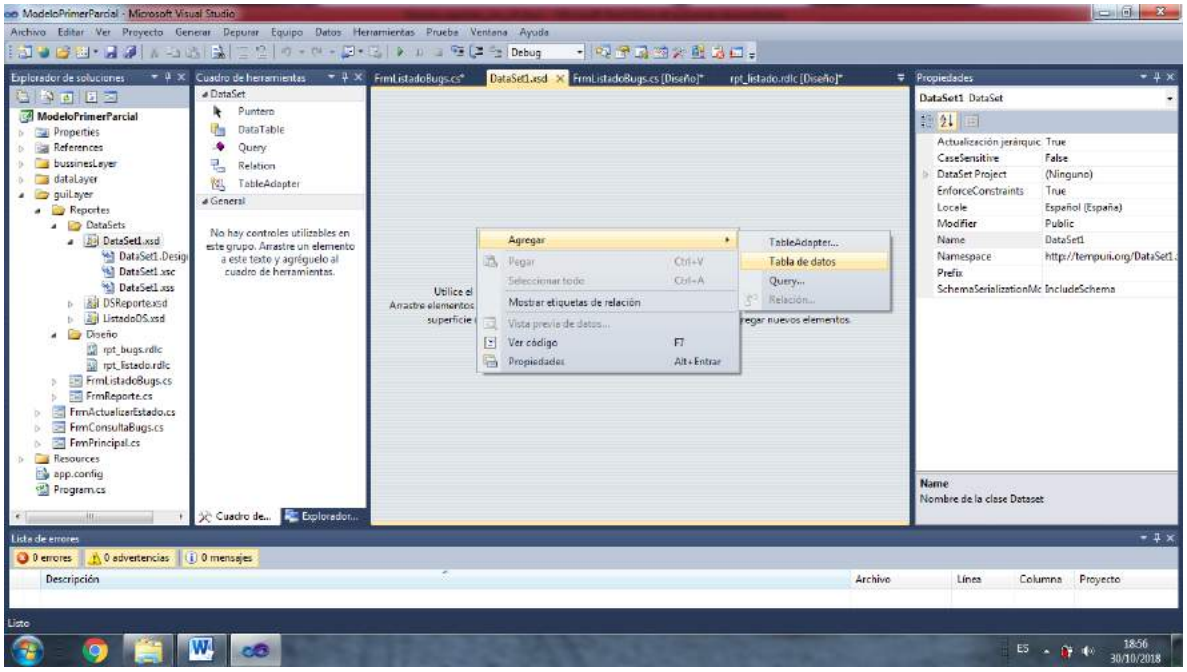

Imagen 28: Elaboración Propia

En este caso la estrategia será crear una tabla en memoria que no existe en nuestro modelo de datos pero que va a contener tantas columnas como campos hayamos definido en una consulta SQL diseñada especialmente para crear el reporte y que podremos solicitar a la capa de datos. La consulta debería contar por Producto y por Estado la cantidad de bugs presentes entre dos fechas parametrizadas, tal como se muestra en la sentencia:

```
string strSql="SELECT t2.descripcion as producto, t3.nombre as estado, COUNT(*) as ctd " +
                    "FROM Bugs t1, Productos t2, Estados t3 " +
                    "WHERE t1.id_producto = t2.id_producto " +
                    "AND t1.id_estado = t3.id_estado " +
                    "AND t1.fecha_alta between @fechaDesde AND @fechaHasta " +
                    "GROUP BY t2.descripcion, t3.nombre";
```

La tabla quedará definida de la siguiente manera:

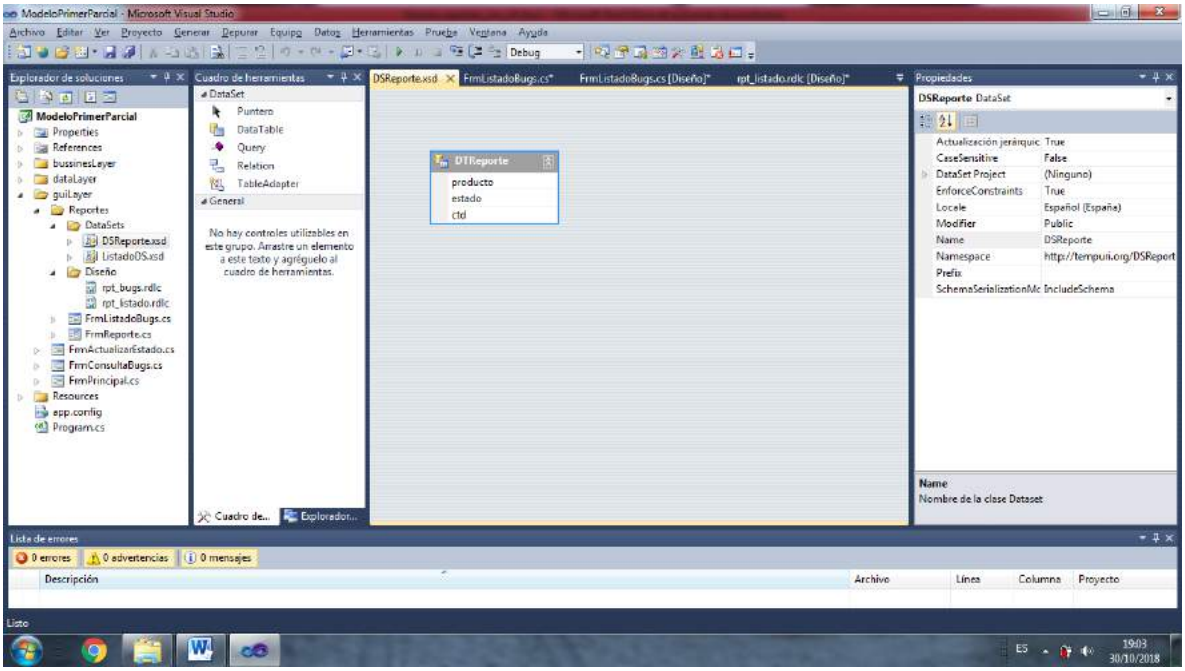

Imagen 29: Elaboración Propia

2. Siguiendo los mismos pasos que en la sección anterior diseñamos el objeto .rdlc. Arrastramos la tabla de contenido y utilizamos el DATASET definido en el punto anterior, resultando:

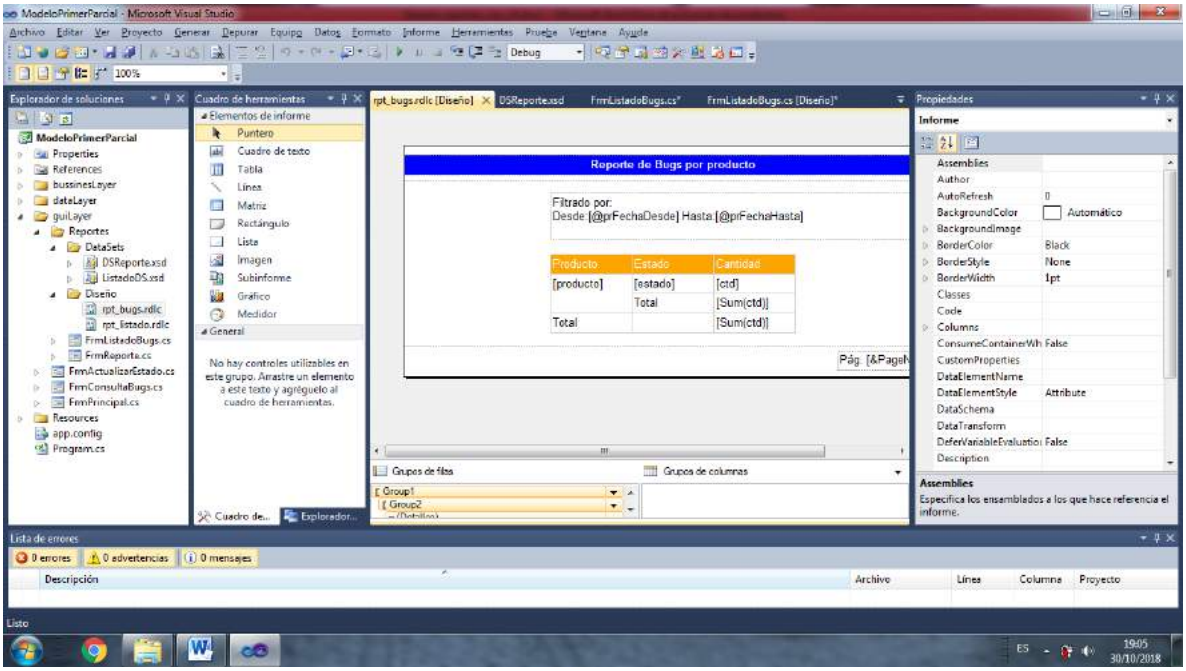

Imagen 30: Elaboración Propia

Para poder armar los grupos (o cortes de control) seleccionamos una fila de la tabla elegimos la opción *Agregar Grupo >>Grupo primario…*

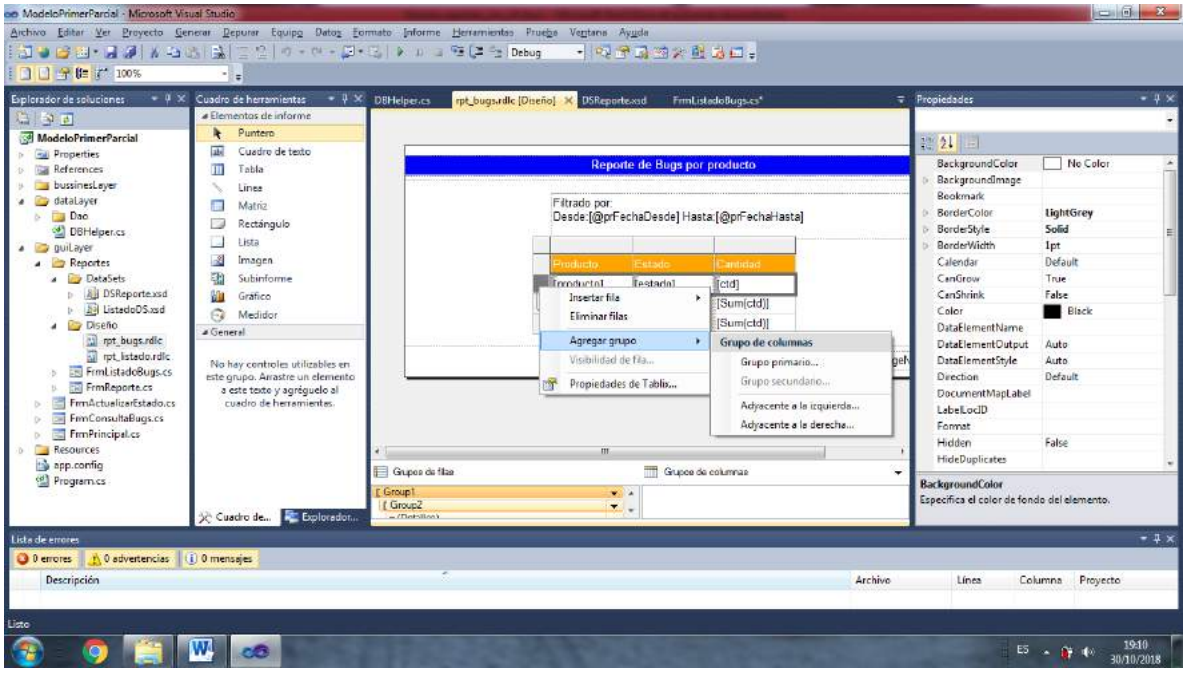

Imagen 31: Elaboración Propia

Una vez creado el grupo primario, siguiendo los mismos pasos podemos crear un Grupo secundario. De esta forma tendremos para cada producto, por estado el total de bugs.

Solo nos resta agregar los totales a los grupos. Para ello, parados sobre un grupo seleccionamos *Agregar Total>> Después de.* Esto nos agrega una fila adicional donde podemos ingresar una **Expresión** con una función integrada Sum() y tendremos los totales del reporte.

3. Por último vamos a enlazar en un formulario un control ReportViewer con un .rdlc del diseño.

Ahora bien, en vez de elegir el origen de datos para el DataSet del reporte lo vamos a hacer por código en un botón **Generar**

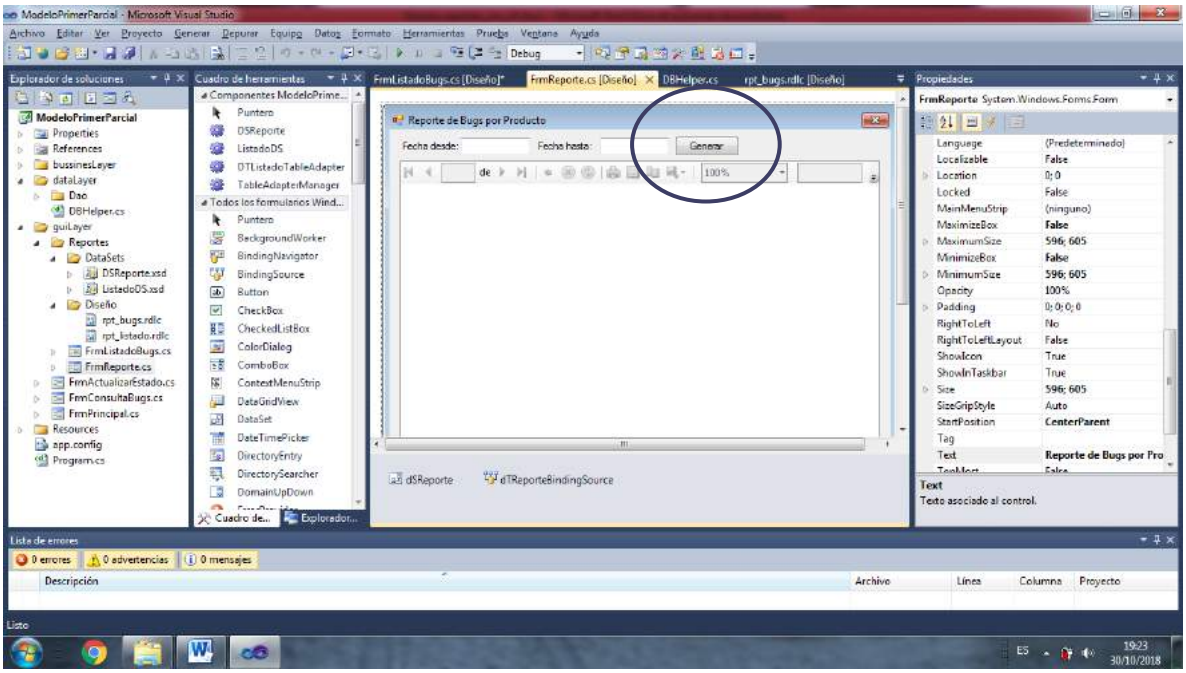

Imagen 32: Elaboración Propia

Registramos un evento de click con el siguiente código:

```
private void btnGenerar_Click(object sender, EventArgs e)
        {
            string strSql = "SELECT t2.nombre as producto, t3.nombre as estado,
COUNT(*) as ctd " +
             "FROM Bugs t1, Productos t2, Estados t3 " +
             "WHERE t1.id_producto = t2.id_producto " +
             "AND t1.id_estado = t3.id_estado " +
             "AND t1.fecha_alta between @fechaDesde AND @fechaHasta " +
             "GROUP BY t2.nombre, t3.nombre";
            // Dictionary: Representa una colección de claves y valores.
            Dictionary<string, object> parametros = new Dictionary<string,
object>();
            DateTime fechaDesde;
            DateTime fechaHasta;
            if (DateTime.TryParse(txtFechaDesde.Text, out fechaDesde) &&
                DateTime.TryParse(txtFechaHasta.Text, out fechaHasta))
            {
                parametros.Add("fechaDesde", txtFechaDesde.Text);
                parametros.Add("fechaHasta", txtFechaHasta.Text);
```

```
                    rpvBugs.LocalReport.SetParameters(new ReportParameter[] { new
ReportParameter("prFechaDesde", txtFechaDesde.Text), new ReportParameter("prFechaHasta",
txtFechaHasta.Text) });

                    //DATASOURCE

                    rpvBugs.LocalReport.DataSources.Clear();

                    rpvBugs.LocalReport.DataSources.Add(new
ReportDataSource("DSReporte", DataManager.GetInstance().ConsultaSQL(strSql, parametros)));

                    rpvBugs.RefreshReport();

                }

            }
```

Básicamente lo que estamos diciendo es:

- Limpiar los DataSources del reporte
- Agregar un nuevo DataSource y asociar el "`DSReporte`" con un datatable devuelto por nuestro `DataManager`.
- Por último llamamos al `RefreshReport()` para ejecutar el reporte.

Importante:

Mediante la línea:

```
rpvBugs.LocalReport.SetParameters(new ReportParameter []{ new
ReportParameter("prFechaDesde", txtFechaDesde.Text), new ReportParameter("prFechaHasta",
txtFechaHasta.Text) });
```

Estamos pasando al diseño del reporte los parámetros "prFechaHasta" y "prFechaDesde" con los valores de las fechas usadas como filtro de consulta.

BIBLIOGRAFÍA

Bishop, P. (1992) Fundamentos de Informática. Anaya.

Brookshear, G. (1994) Computer Sciense: An Overview. Benjamin/Cummings.

De Miguel, P. (1994) Fundamentos de los Computadores. Paraninfo.

Joyanes, L. (1990) Problemas de Metodología de la Programación. McGraw Hill.

Joyanes, L. (1993) Fundamentos de Programación: Algoritmos y Estructura de Datos. McGraw Hill.

Norton, P. (1995) Introducción a la Computación. McGraw Hill.

Prieto, P. Lloris A. y Torres J.C. (1989) Introducción a la Informática. McGraw Hill.

Tucker, A. Bradley, W. Cupper, R y Garnick, D (1994) Fundamentos de Informática (Lógica, resolución de problemas, programas y computadoras). McGraw Hill.

ANEXO: INSTALACIÓN DE COMPLEMENTOS VISUAL STUDIO

Instalación de Microsoft RDLC Report Designer

1. Abrimos Visual Studio, de la barra de herramientas seleccionamos **Extensiones**. Clic en la opción de **Administrador de Extensiones**.

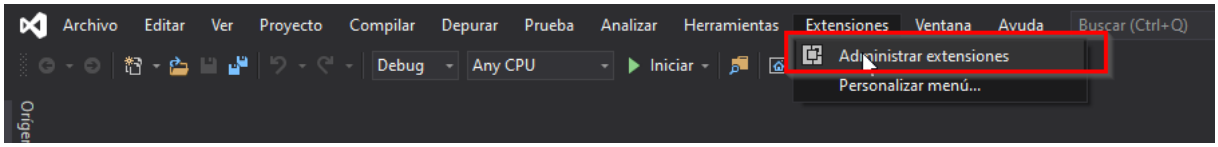

Imagen 33: Elaboración Propia

2. De las secciones seleccionamos **En línea**, en el buscador escribimos **Report**, descargamos las dos primeras extensiones.

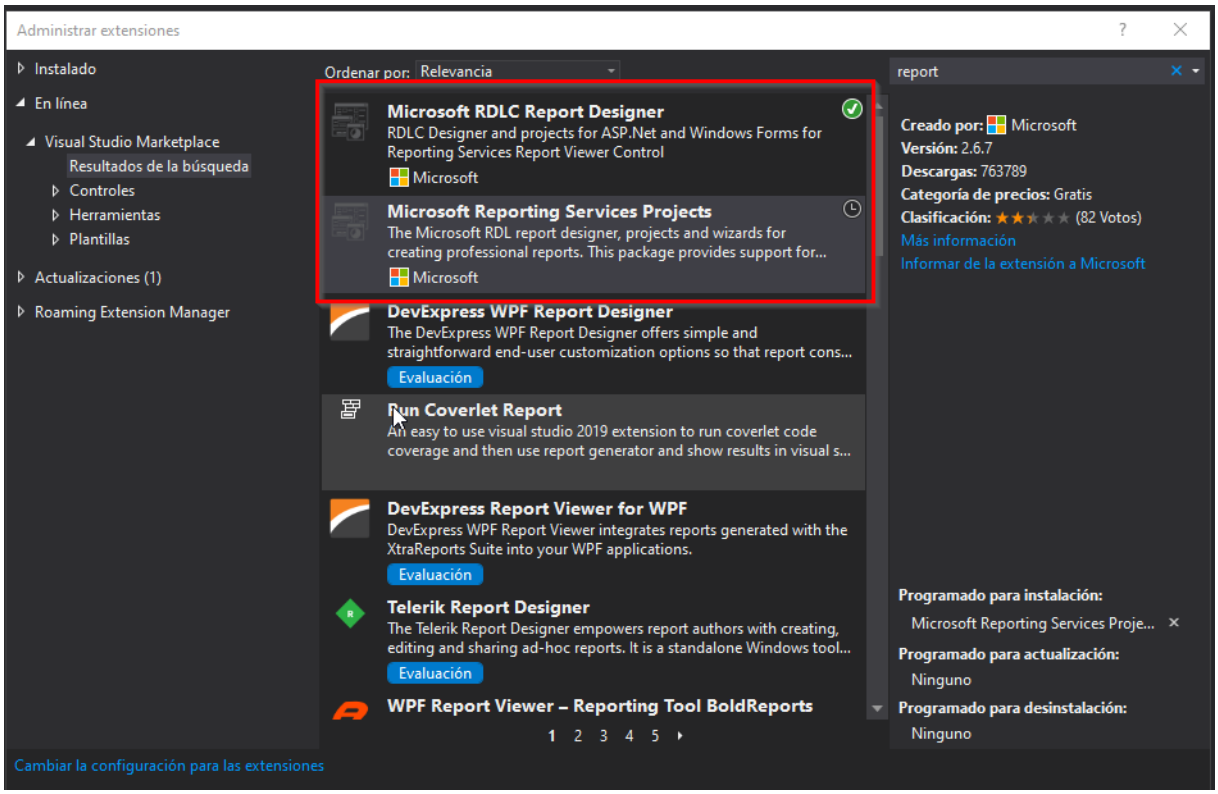

Imagen 34: Elaboración Propia

Nota: Es necesario cerrar Visual Studio para completar la instalación de las extensiones.

Instalación de Paquete NuGet para el control ReportViewer

Para utilizar el control ReportViewer es necesario instalar un Paquete Nuget de la siguiente forma:

1. Abrimos Visual Studio, de la barra de herramientas seleccionamos **Herramientas**. Clic en la opción de **Administrador de paquetes NuGet** y luego en **Administrar paquetes NuGet para la solución…**

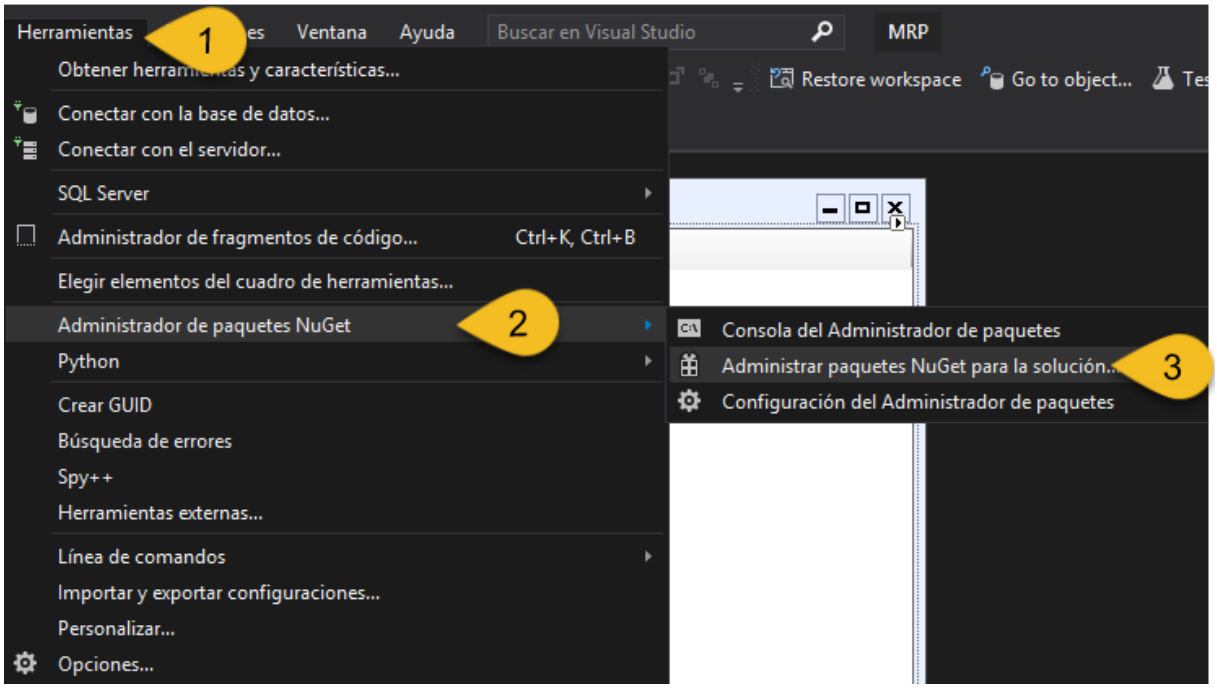

Imagen 35: Elaboración Propia

2. Seleccionamos la siguiente versión: **Microsoft.ReportingServices.ReportViewerControl.Winforms**

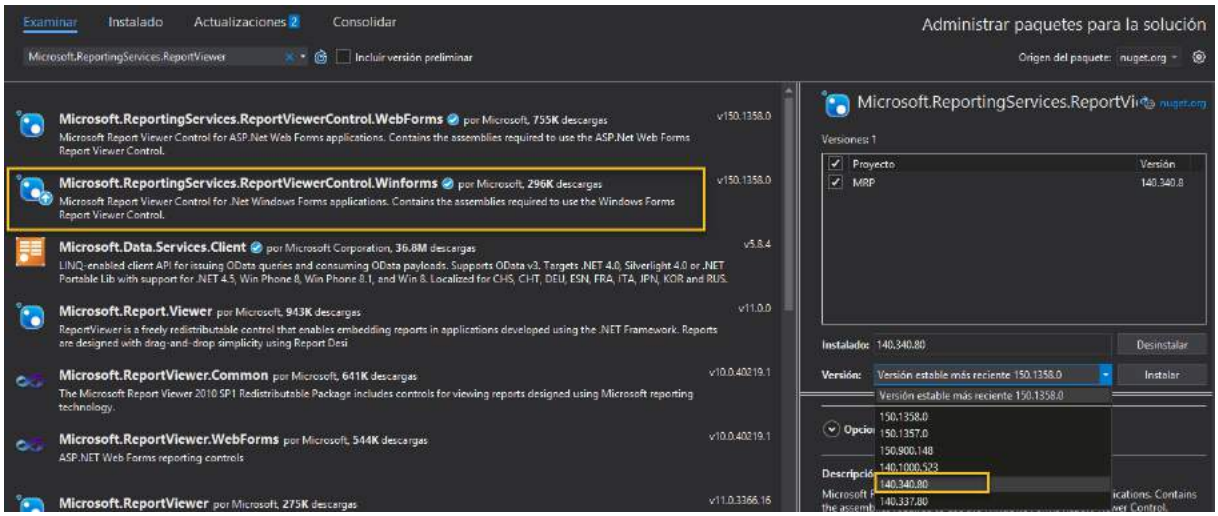

Imagen 36: Elaboración Propia

3. Una vez que se instala vamos al formulario y hacemos click derecho sobre el cuadro de herramientas

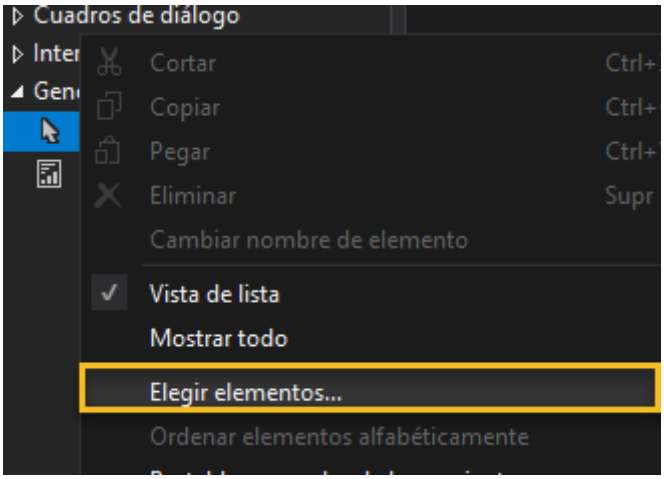

Imagen 37: Elaboración Propia

4. En la ventana que aparece, click en **examinar** y vamos a la ruta del proyecto, ingresamos a la carpeta de la extensión que se instala:

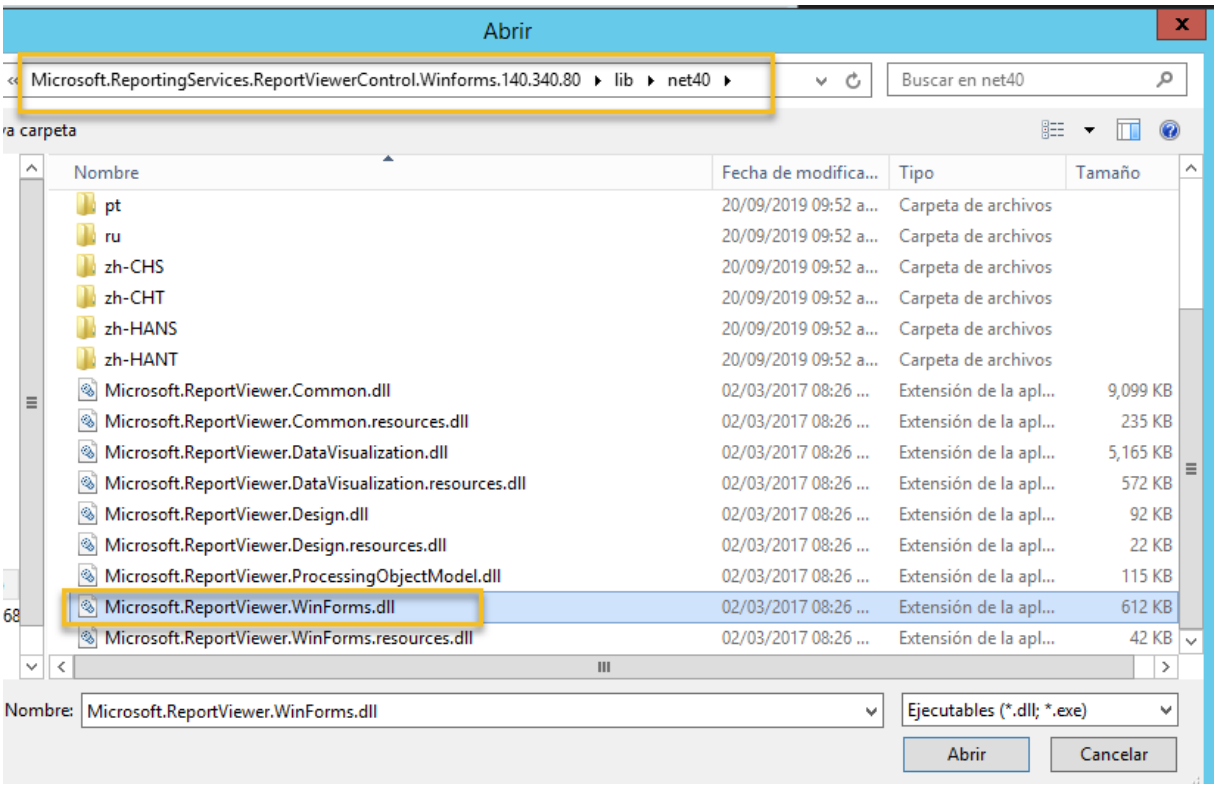

Imagen 38: Elaboración Propia

5. El control se cargará, seleccionamos y aceptamos:

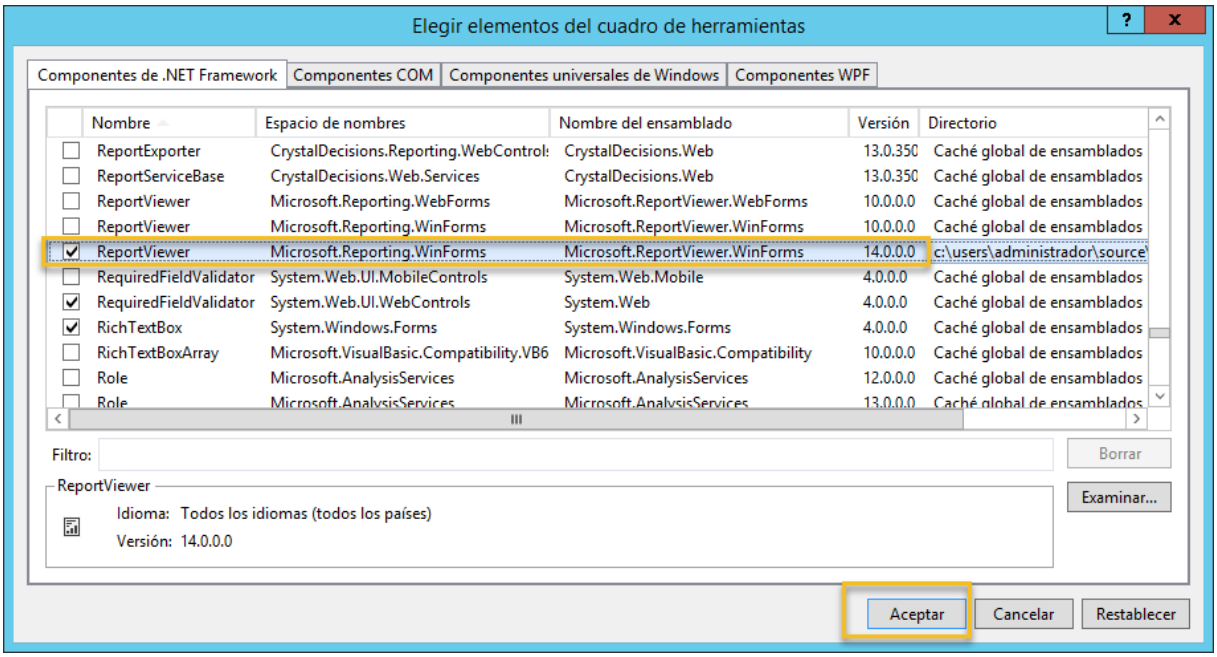

Imagen 39: Elaboración Propia

6. El control aparecerá en tu cuadro de herramientas y podremos arrastrarlo al formulario.

Nota: Es necesario cerrar Visual Studio y volver a abrir para visualizar el control en el Cuadro de Herramientas.