

TAREA: CLASE 10

TAREAS CAPACITACIÓN C#

 **Ezequiel Remus**
ezequielremus@gmail.com

13 de abril de 2023

RESUMEN

En este apunte se encuentran alojadas las resoluciones a cada ejercicio de la Tarea de la clase 10 de la Capacitación en C#. En este caso, nos encontraremos trabajando con *Entity Framework*. Continuamos modificando el proyecto StockApp con el objetivo de que las clases Deposito y Stock sean manejadas totalmente con las funcionalidades provistas por *Entity Framework*.

1. Agrego Foreign Keys en SQL Server

1.1. Enunciado

Agregar las Foreign Keys a las tabla Stock, para relacionarla con la tabla Articulos y Depositos. Deberían quedar como muestra la imagen

1.2. Solución

Lo que primero debemos hacer es abrir SQL Server y mediante Diagramas de Bases de Datos relacionar las tablas de Stock, Deposito y Articulo mediante **Foreing Keys**. Apoyandonos en los atributos de los Id de atributo y deposito de la tabla de stock y manteniendo el clic arrastramos el mouse a la tabla respectiva. Nos debe quedar como en la Figura

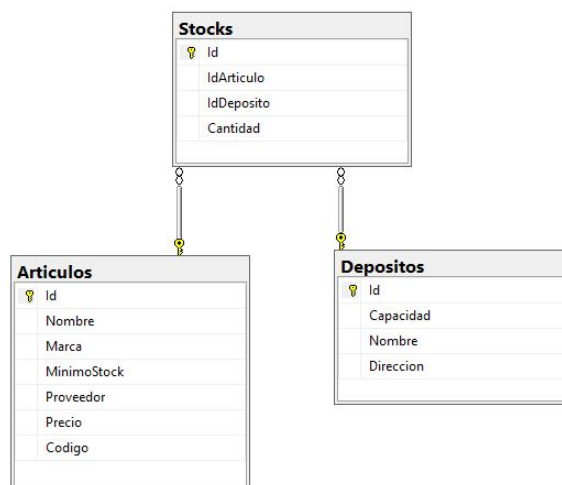


Figura 1: Tabla de Relaciones

2. Ingeniería Inversa

2.1. Enunciado

Usando Entity Framework hacer “Ingeniería inversa”, actualizar las clase creadas por entity framework agregando la clase Stock .

2.2. Solución

Haciendo los mismos pasos que seguimos en la *Tarea: Clase 9*, hacemos ingeniería inversa sobre el proyecto CodigoComun

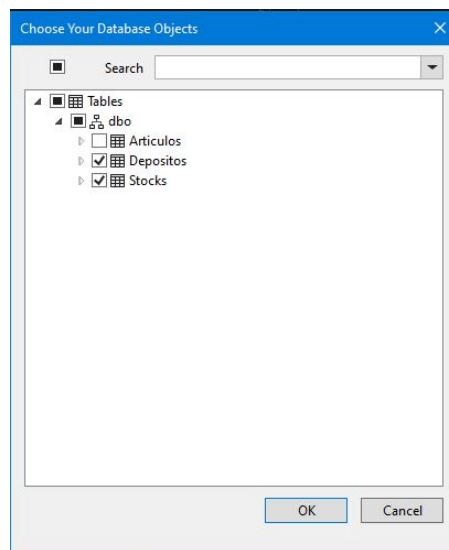


Figura 2: Tablas a las cuales le aplicamos Ingeniería Inversa

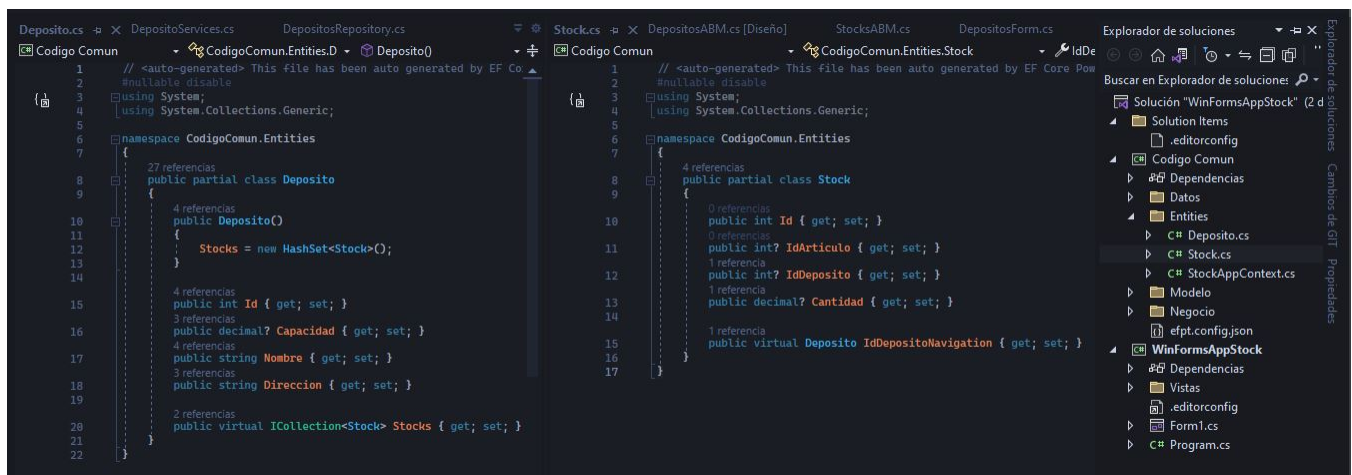


Figura 3: Clases creadas por Entiti y árbol creado

3. Método Actualizar Deposito

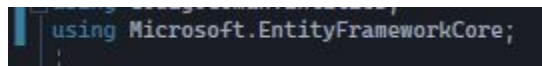
3.1. Enunciado

Agregar el método Actualizar Deposito que no se agrego en la tarea anterior, Repository y Services. También modificarlo en el proyecto de Winform.

Enviar captura de pantalla del árbol de visual studio donde se vean estas clases.

3.2. Solución

Ahora, utilizaremos la clase **Microsoft.EntityFrameworkCore** para poder modificar un deposito. Lo que primero haremos es colocar el siguiente using



```
using Microsoft.EntityFrameworkCore;
```

Figura 4: Using

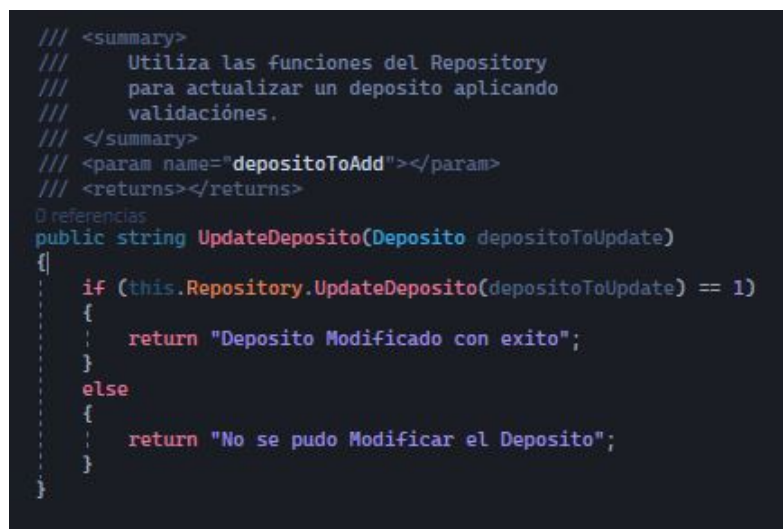
Luego, este nos permitira usar EntityState, para modificar la base de datos. El codigo del método de Actualizar nos quedara como se ve en la figura 5



```
/// <summary>
///     Utiliza Entity Framework para
///     modificar el deposito pasado
/// </summary>
/// <param name="depositoToUpdate"></param>
/// <returns></returns>
0 referencias
public int UpdateDeposito(Deposito depositoToUpdate)
{
    db.Entry(depositoToUpdate).State = EntityState.Modified;
    return db.SaveChanges();
}
```

Figura 5: UpdateDeposito() en DepositosRepository

Por ultimo, no debemos olvidar de crear el servicio



```
/// <summary>
///     Utiliza las funciones del Repository
///     para actualizar un deposito aplicando
///     validaciones.
/// </summary>
/// <param name="depositoToAdd"></param>
/// <returns></returns>
0 referencias
public string UpdateDeposito(Deposito depositoToUpdate)
{
    if (this.Repository.UpdateDeposito(depositoToUpdate) == 1)
    {
        return "Deposito Modificado con exito";
    }
    else
    {
        return "No se pudo Modificar el Deposito";
    }
}
```

Figura 6: UpdateDeposito() en DepositosServices

4. Capa de Datos: StockRepository

4.1. Enunciado

Crear en la Capa de Datos un “StockRepository” y el mismo debe implementar los siguientes métodos utilizando las clases de entity framework generadas en el punto 1

- 5.3GetTodosStocks()
- 5.4GetStockPorId()
- 5.5AddStock()
- 4.6Actualizar Stock()
- 4.7EliminarDeposito()

Enviar captura de pantalla del repository donde se vean estos métodos implementados

4.2. Solución

Este punto es básicamente hacer lo que fuimos haciendo en clase

4.3. GetTodosStocks()

```
/* GETTERS */  
/// <summary>  
/// Toma los todos los stocks de la DB  
/// </summary>  
/// <returns>La lista de Stock</returns>  
0 referencias  
public List<Stock> GetTodosLosStocks()  
{  
    List<Stock> stocksADevolver = new List<Stock>();  
    stocksADevolver = this.db.Stocks.ToList();  
  
    return stocksADevolver;  
}
```

Figura 7: Código del método GetTodosStocks ()

4.4. GetStockPorId()

```
/// <summary>  
/// Toma el stock pasado por id  
/// </summary>  
/// <param name="idStock"></param>  
/// <returns></returns>  
1 referencia  
public Stock GetStockPorId(int idStock)  
{  
    // warning disable CS8600 // Se va a convertir un literal nulo o un posible val  
    Stock stockADevolver = this.db.Stocks.Where(  
        p => p.Id == idStock).FirstOrDefault();  
    // warning disable CS8603 // Posible tipo de valor devuelto de referencia nulo  
    return stockADevolver;  
}
```

Figura 8: Código del método GetStockPorId()

4.5. AddStock()

```
/* A.B.M */  
  
/// <summary>  
///     Utiliza Entity Framework para  
///     guardar el stock en la base de datos  
/// </summary>  
/// <param name="stockToAdd"></param>  
/// <returns></returns>  
0 referencias  
public int AddStock(Stock stockToAdd)  
{  
    this.db.Stocks.Add(stockToAdd);  
    int r = db.SaveChanges();  
    return r;  
}
```

Figura 9: Código del método AddStock()

4.6. Actualizar Stock()

```
/// <summary>  
///     Utiliza Entity Framework para  
///     eliminar el Stock de la base de datos  
/// </summary>  
/// <param name="idStockToDelete"></param>  
/// <returns></returns>  
0 referencias  
public int DeleteStock(int idStockToDelete)  
{  
    // Obtengo al deposito  
    Stock stockToDelete = this.GetStockPorId(idStockToDelete);  
    // lo borro de la DB  
    this.db.Stocks.Remove(stockToDelete);  
  
    return this.db.SaveChanges();  
}
```

Figura 10: Código del método Actualizar Stock()

4.7. EliminarDeposito()

```
/// <summary>  
///     Utiliza Entity Framework para  
///     modificar el stock pasado  
/// </summary>  
/// <param name="stockToUpdate"></param>  
/// <returns></returns>  
0 referencias  
public int UpdateStock(Stock stockToUpdate)  
{  
    db.Entry(stockToUpdate).State = EntityState.Modified;  
    return db.SaveChanges();  
}
```

Figura 11: Código del método EliminarDeposito()

5. Capa de Negocios: StockServices

5.1. Enunciado

Crear un **StockServices** que utilizando el repository del punto anterior implemente sus funcionalidades.

Enviar Captura de esta Clases donde se vean los metodos

5.2. Solución

Este punto es básicamente hacer lo que fuimos haciendo en clase

5.3. GetTodosStocks()

```
/* ATRIBUTOS */
private StocksRepository Repository = new StocksRepository();

/* METODOS */
/* GETTERS */

/// <summary>
/// Toma el Stock por el id pasado
/// </summary>
/// <param name="idStock"></param>
/// <returns></returns>
#pragma warning disable IDE0022 // Usar cuerpo del bloque para el método
0 referencias
public Stock GetDeposito(int idStock) => Repository.GetStockPorId(idStock);
```

Figura 12: Código del método GetTodosStocks () en StockServices

5.4. GetStockPorId()

```
/// <summary>
/// Toma una lista de Stock de la DB
/// </summary>
/// <returns></returns>
#pragma warning disable IDE0022 // Usar cuerpo del bloque para el método
0 referencias
public List<Stock> GetTodosLosStocks() => Repository.GetTodosLosStocks();
```

Figura 13: Código del método GetStockPorId() en StockServices

5.5. AddStock()

5.6. Actualizar Stock()

5.7. EliminarDeposito()

```
/* A.B.M */  
/// <summary>  
/// Utiliza las funciones del Repository  
/// para agregar un deposito aplicando  
/// validaciones.  
/// </summary>  
/// <param name="stockToAdd"></param>  
/// <returns></returns>  
0 referencias  
public string AddStock(Stock stockToAdd)  
{  
    if (this.Repository.AddStock(stockToAdd) == 1)  
    {  
        return "Stock Agregado con éxito";  
    }  
    else  
    {  
        return "No se pudo agregar el Stock";  
    }  
}
```

Figura 14: Código del método AddStock() en StockServices

```
/// <summary>  
/// Utiliza las funciones del Repository  
/// para eliminar un deposito.  
/// </summary>  
/// <param name="idStockToDelete"></param>  
/// <returns></returns>  
0 referencias  
public string DeleteStock(int idStockToDelete)  
{  
    // Agrego el deposito en la base  
    if (this.Repository.DeleteStock(idStockToDelete) == 1)  
    {  
        return "Stock Eliminado con éxito";  
    }  
    else  
    {  
        return "No se pudo eliminar el Stock";  
    }  
}
```

Figura 15: Código del método Actualizar Stock() en StockServices

```
/// <summary>  
/// Utiliza las funciones del Repository  
/// para actualizar un deposito aplicando  
/// validaciones.  
/// </summary>  
/// <param name="stockToAdd"></param>  
/// <returns></returns>  
0 referencias  
public string UpdateStock(Stock stockToUpdate)  
{  
    if (this.Repository.UpdateStock(stockToUpdate) == 1)  
    {  
        return "Stock Modificado con éxito";  
    }  
    else  
    {  
        return "No se pudo Modificar el Stock";  
    }  
}
```

Figura 16: Código del método DeleteStock() en StockServices

6. Borramos las clases del Models

6.1. Enunciado

Eliminar a Clase **Deposito** y **Stock** de la carpeta Modelo. Deberían solo quedar en la carpeta Entities.

6.2. Solución

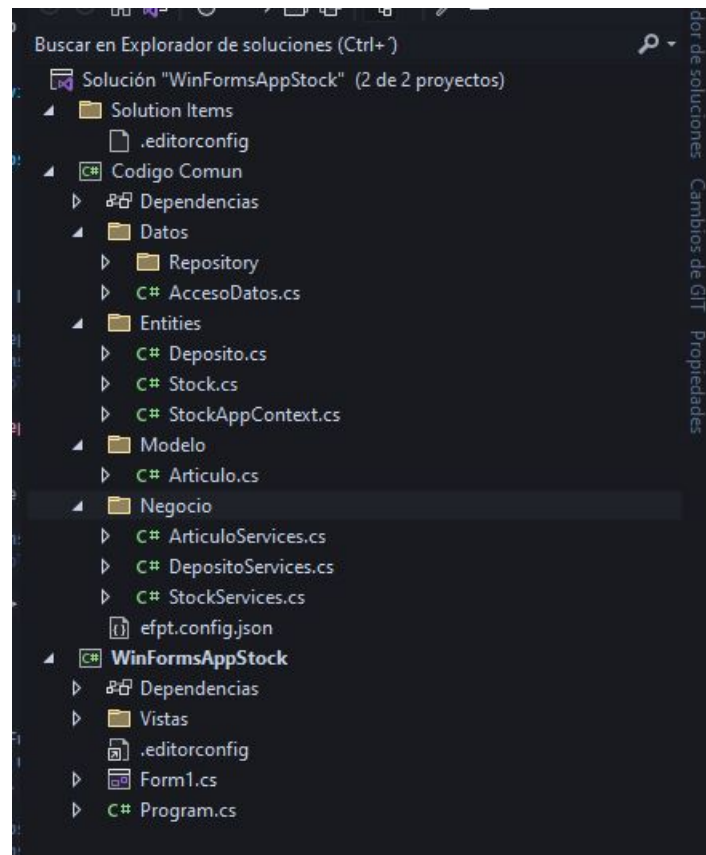


Figura 17: Sacamos las clases Deposito y Stock del Models

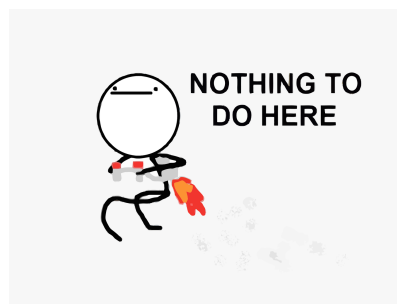


Figura 18:

7. Capa de Presentación: Proyecto winform

7.1. Enunciado

Modificar en la capa de presentación (Proyecto winform) al realizar altas, bajas modificación y los gets de Stock se haga utilizando los services, Idem para deposito que faltaba el metodo actualizar

Enviar para el un print de pantalla del código utilizando el services y un print donde se vea que el alta/baja/modificacion de stock sigue funcionando con esta nueva funcionalidad. Idem para la grilla donde se muestra el stock.

7.2. Solución

Ahora si, llego la hora de arremangarse y ponerse a laburar, Entity no iba a hacer todo el trabajo por nosotros. Ahora tenemos que modificar las Vistas para que todo funcione como lo hacia antes.

Primero, vamos a modificar el **ModificarDeposito()** de la clase **DepositoABM()**

```

/// <summary>
///     Modifica los datos del deposito pasado
///     segun los textBox del formulario
/// </summary>
1 referencia
private void ModificarDeposito()
{
    DepositoServices depositoServices = new DepositoServices();
    Deposito depositoAModificar = new Deposito();
    //Deposito depositoAux = new Deposito();

    depositoAModificar.Id = Convert.ToInt32(txtId.Text);
    depositoAModificar.Capacidad = Convert.ToDecimal(txtCapacidad.Text);
    depositoAModificar.Nombre = txtNombre.Text;
    depositoAModificar.Direccion = txtDireccion.Text;
    string mensaje = depositoServices.UpdateDeposito(depositoAModificar);

    if (mensaje == "Deposito Modificado con exito")
    {
        MessageBox.Show(mensaje, "Operación Exitosa");
        this.Close();
    }
    else
    {
        MessageBox.Show(mensaje, "Hubo un problema");
    }
}

```

Figura 19: DepositoABM()/ModificarDeposito()

Ahora viene la parte ardua y es modificar todas las clases de Stock.

Lo que primero que voy a hacer es declarar todos los atributos en **StockABM.cs** para no ir repitiendo código innecesario.

```

namespace WinFormsAppStock.Vistas
{
    7 referencias
    public partial class StocksABM : Form
    {
        /*
        * ATRIBUTOS
        */
        private ArticuloServices articuloServices = new ArticuloServices();
        private StockServices stockServices = new StockServices();
        private DepositoServices depositoServices = new DepositoServices();

        3 referencias
        private bool EstoyModificando { get; set; }

        private Stock stockAux = new Stock();
        private Articulo articuloAux = new Articulo();
        private Deposito depositoAux = new Deposito();
    }
}

```

Figura 20: Atributos

Ahora, en los constructores no tenemos que tocar nada, pero si debemos modificar los métodos que utilizamos para Cargar los datos.

Primero, el que carga los datos en la planilla de ABM cuando queremos modificar. El cambio que hay que hacer es tomar el deposito que se pasa por Id en el **Forms** se haga con **StockServices.cs**.

```

/// <summary>
///     Carga los datos en los textBox y control Box
///     de el Stock pasado por id
/// </summary>
/// <param name="idStockAModificar"></param>
1 referencia
private void CargarDatosStockParaModificar(int idStockAModificar)
{
    stockAux = stockServices.GetDeposito(idStockAModificar);

    txtId.Text = Convert.ToString(idStockAModificar);
    txtCantidad.Text = stockAux.Cantidad.ToString();
    cbArticulo.Text = stockAux.IdArticulo.ToString();
    cbDeposito.Text = stockAux.IdDeposito.ToString();
}

```

Figura 21: Código del método CargarDatosStockParaModificar()

Ahora, recordemos que para poder cargar los ControlBox del stock teníamos que poder pasarle una lista de los depositos y los atributos. Este metodo lo habiamos tenido que modificar ya en la tarea pasada, cuando realizamos el GetTodosLosDepositos() en la capa de negocios para el deposito.

```

/// <summary>
///     Carga los datos de los articulos y los depositos en los control box.
/// </summary>
2 referencias
private void CargarControlBox()
{
    List<Deposito> depositosAux = new List<Deposito>();
    List<Articulo> articulosAux = new List<Articulo>();

    articulosAux = articuloServices.GetTodosPorID();
    depositosAux = depositoServices.GetTodosLosDepositos();

    cbArticulo.DataSource = new BindingSource(articulosAux, null);
    cbArticulo.DisplayMember = "Nombre";
    cbArticulo.ValueMember = "Id";
    cbDeposito.DataSource = new BindingSource(depositosAux, null);
    cbDeposito.DisplayMember = "Nombre";
    cbDeposito.ValueMember = "Id";
}

```

Figura 22: Código del método CargarControlBox()

Ahora, para poder agregar o modificar el stock, tambien debemos hacerlo desde el Servicio. Además, devemos modificar las condiciones pasandole el mensaje como la variable de comparación.

```

/// <summary>
///     Agrega los datos del formulario
///     a la base de datos
/// </summary>
1 referencia
private void AgregarStock()
{
    var articuloSeleccionado = cbArticulo.SelectedItem;
    var depositoSeleccionado = cbDeposito.SelectedItem;

    articuloAux = (Articulo)articuloSeleccionado;
    depositoAux = (Deposito)depositoSeleccionado;

    stockAux.Cantidad = Convert.ToDecimal(txtCantidad.Text);
    stockAux.IdArticulo = articuloAux.Id;
    stockAux.IdDeposito = depositoAux.Id;

    string mensaje = stockServices.AddStock(stockAux);

    if (mensaje == "Stock Agregado con exito")
    {
        MessageBox.Show(mensaje, "Operación Exitosa");
        this.Close();
    }
    else
    {
        MessageBox.Show(mensaje, "Hubo un problema");
    }
}

```

Figura 23: Código del método AgregarStock()

```

/// <summary>
///     Modifica los datos del stock pasado
///     segun los textBox y control box del formulario
/// </summary>
1 referencia
private void ModificarStock()
{
    var articuloSeleccionado = cbArticulo.SelectedItem;
    var depositoSeleccionado = cbDeposito.SelectedItem;

    articuloAux = (Articulo)articuloSeleccionado;
    depositoAux = (Deposito)depositoSeleccionado;

    stockAux.Id = Convert.ToInt32(txtId.Text);
    stockAux.Cantidad = Convert.ToDecimal(txtCantidad.Text);
    stockAux.IdArticulo = Convert.ToInt32(articuloAux.Id);
    stockAux.IdDeposito = Convert.ToInt32(depositoAux.Id);

    string mensaje = stockServices.UpdateStock(stockAux);

    if (mensaje == "Stock Modificado con exito")
    {
        MessageBox.Show(mensaje, "Operación Exitosa");
        this.Close();
    }
    else
    {
        MessageBox.Show(mensaje, "Hubo un problema");
    }
}

```

Figura 24: Código del método ModificarStock()

Por ultimo, debemos ir al StocksForm.cs y modificar la carga del GridView para que se haga mediante el servicio y lo mismo para la eliminación de un stock

```

/*****
 * METODOS
 *****/
/// <summary>
///     Carga los datos de la base de datos de stock
///     en el GridView
/// </summary>
1 referencia
private void CargarStocks()
{
    Stock stockAux = new Stock();
    List<Stock> StockAMostrar = stockServices.GetTodosLosStocks();

    gvStocks.Columns.Add("0", "id");
    gvStocks.Columns.Add("1", "Articulo");
    gvStocks.Columns.Add("2", "Deposito");
    gvStocks.Columns.Add("3", "Cantidad");
    for (int i = 1; i <= StockAMostrar.Count; i++)
    {
        int n = gvStocks.Rows.Add();
        gvStocks.Rows[n].Cells[0].Value = StockAMostrar[n].Id.ToString();
        gvStocks.Rows[n].Cells[1].Value = StockAMostrar[n].IdArticulo.ToString();
        gvStocks.Rows[n].Cells[2].Value = StockAMostrar[n].Id.ToString();
        gvStocks.Rows[n].Cells[3].Value = StockAMostrar[n].Cantidad.ToString();
    }

    // Si borramos el codigo anterior y dejamos solo esta linea funciona pero
    // los datos del articulo y el deposito no se toman correctamente
    //gvStocks.DataSource = StockAMostrar;
}

```

Figura 25: Código del método CargarStocks()

```

/*****
 * BOTONES
 *****/
/// <summary>
///     Elimina un Stock una vez indicado el id en
///     el textBox
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 referencia
private void btnEliminarStock_Click(object sender, EventArgs e)
{
    if (string.IsNullOrEmpty(txtIdStock.Text))
    {
        MessageBox.Show("Por favor, ingrese un numero de Id correspondiente a un Stock.", "Cuidado!");
    }
    else
    {
        int idStockAEliminar = Convert.ToInt32(txtIdStock.Text);
        string mensaje = stockServices.DeleteStock(idStockAEliminar);

        if (mensaje == "Stock Eliminado con éxito")
        {
            MessageBox.Show(mensaje, "Operación Exitosa");
        }
        else
        {
            MessageBox.Show(mensaje, "Hubo un Problema");
        }
    }
}

```

Figura 26: Código del método btnEliminarStock_Click()

Ahora, hagamos las pruebas respectivas

7.2.1. Modifico un deposito

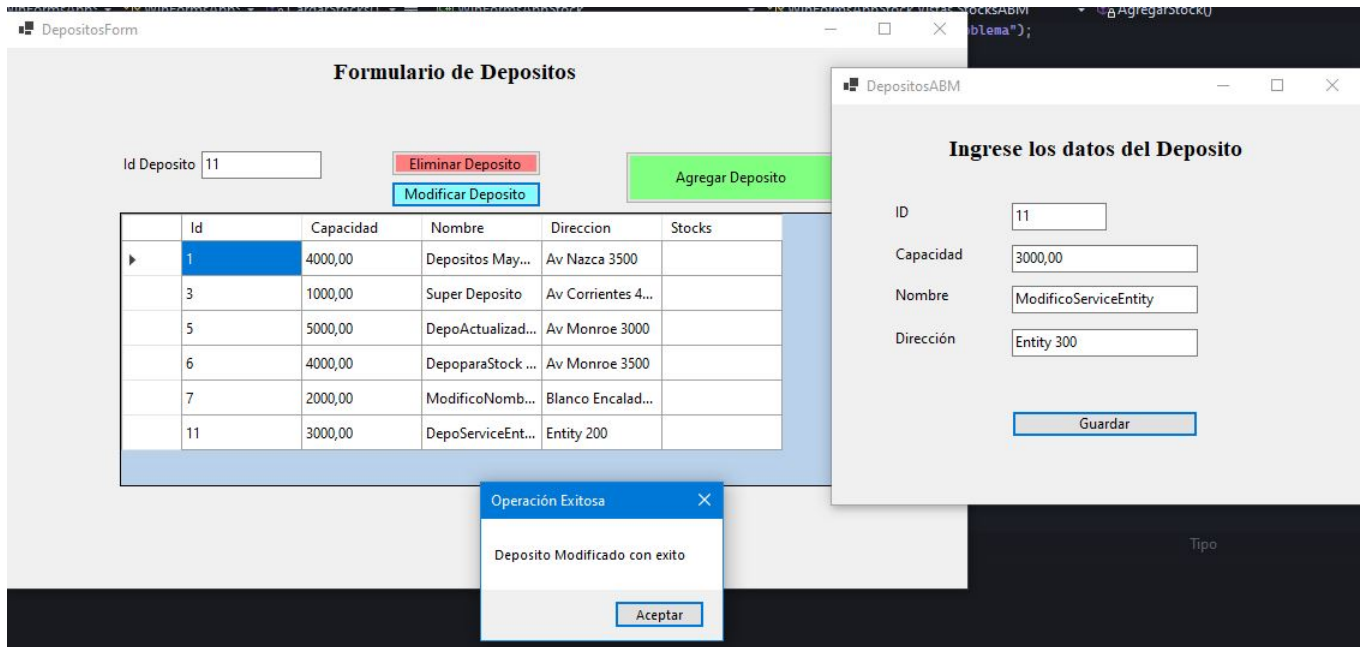


Figura 27: Antes

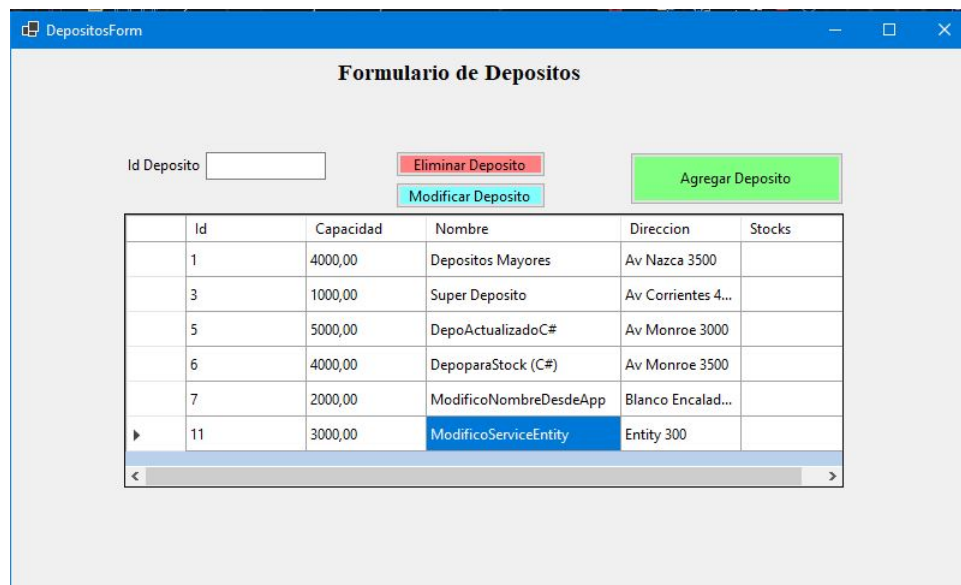


Figura 28: Despues

7.2.2. Agrego un stock

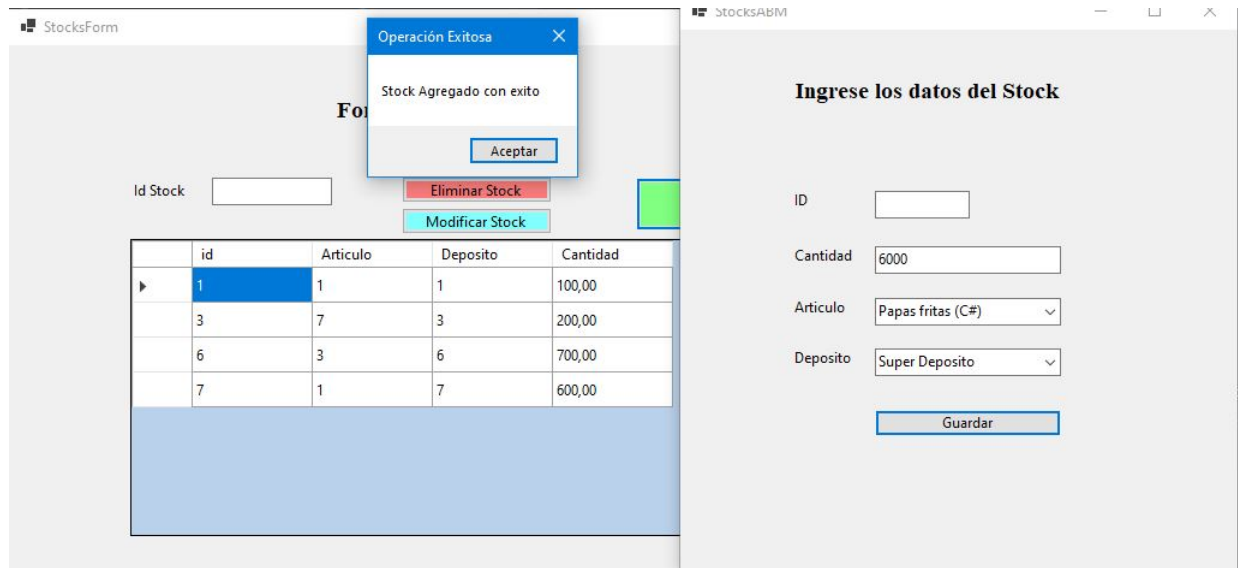


Figura 29: Antes

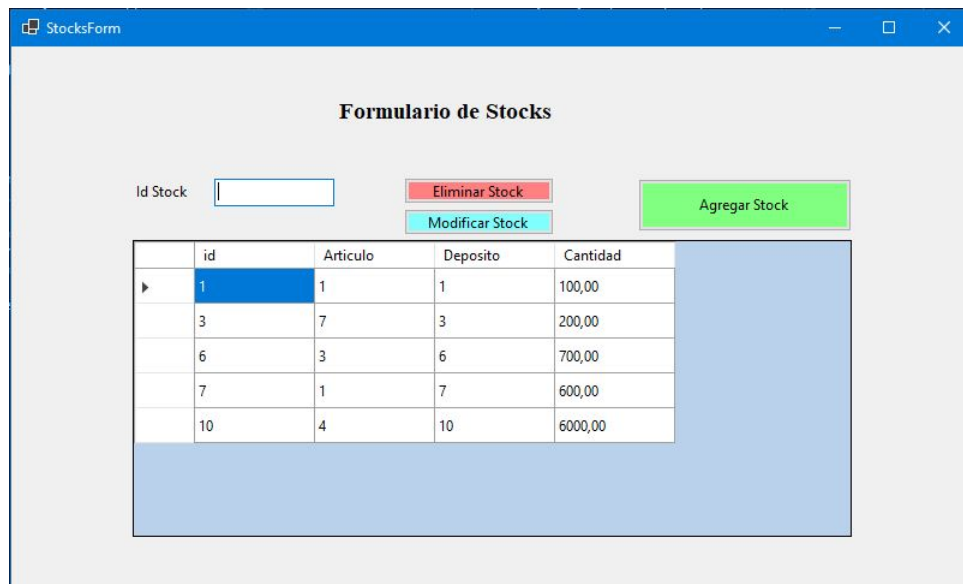


Figura 30: Despues

7.2.3. Modifico un stock

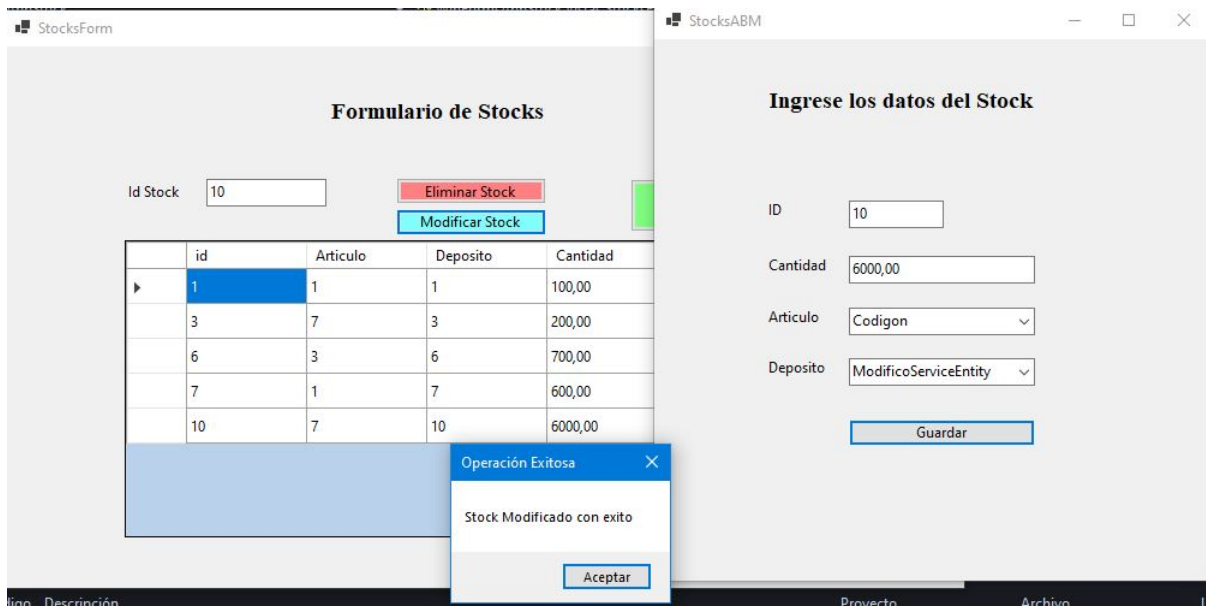


Figura 31: Antes

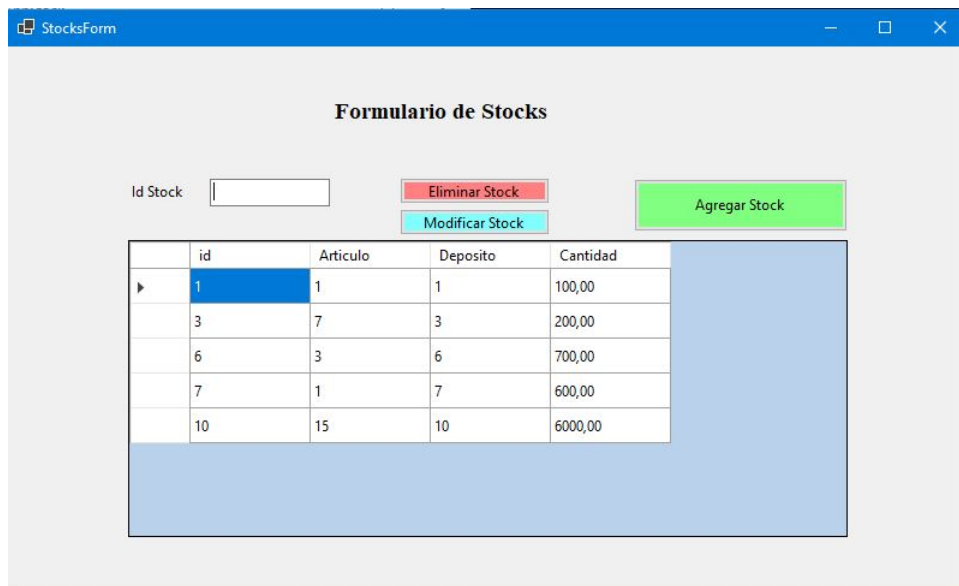


Figura 32: Despues

7.2.4. Elimino un stock

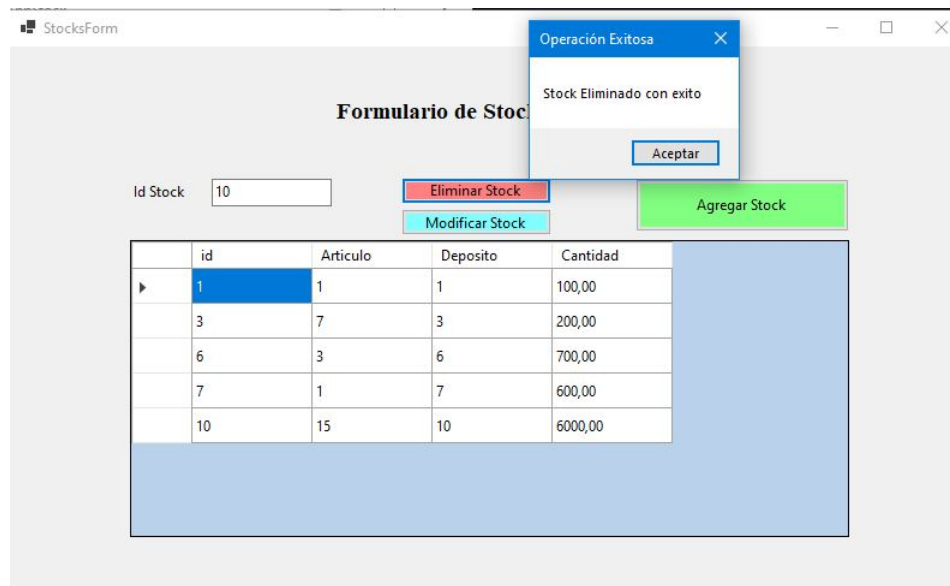


Figura 33: Antes

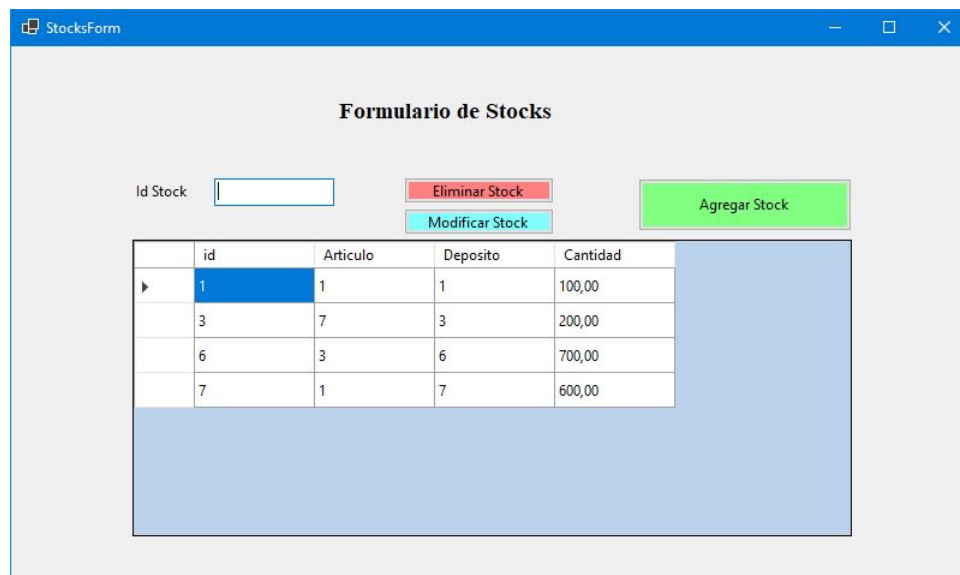


Figura 34: Despues

8. Referencias

1. Link al Drive con el Código