¡Información Importante antes de empezar!
Dentro de la carpeta TP4 se encuentra el script con toda la información de la Base De Datos: **"Zafferano_Gonzalo_TP4_2C.sql".**
Todos los archivos se generan en la ruta:
Zafferano.Gonzalo.2C.TP4\Vista\bin\Debug\net5.0-windows\RespaldoArchivos

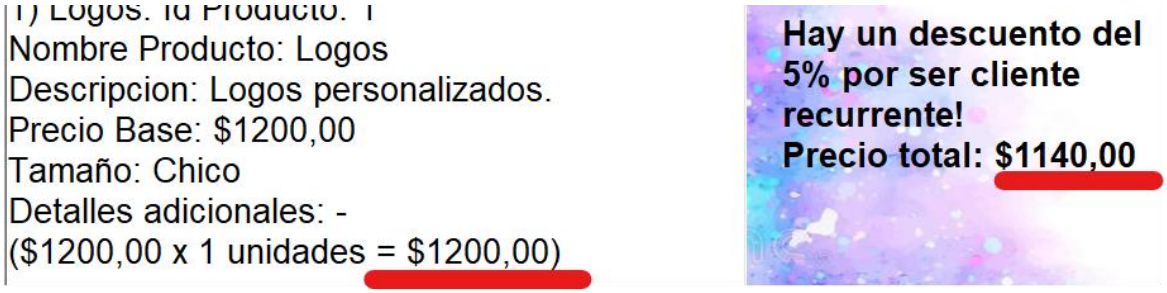
Lo primero será elegir que fuente de información utilizaremos para importar los datos al sistema, teniendo a disposición ***Archivos*** y ***Base de Datos*****:**

Una vez elegida una fuente de información, podemos ***iniciar sesión*** ingresando el **usuario** y **contraseña.**
Jefe: Usuario: **Fguetta** Contraseña: **aaaa**
Administrador: Usuario: **Vero90** Contraseña: **bbbb**
Empleado: Usuario: **Lramos** Contraseña: **cccc**

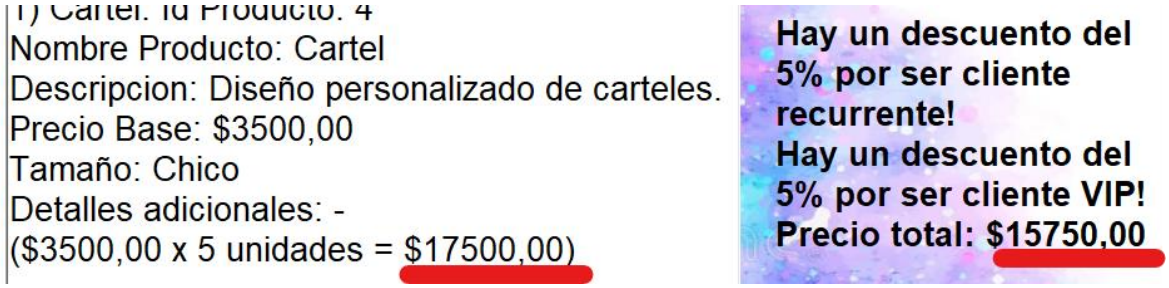
Adicionalmente, para ***agilizar las pruebas***, podemos ***AUTOMATIZAR*** el ingreso, utilizando los botones en la derecha para ingresar como un **Usuario, Administrador o Jefe.**
-El ***usuario*** podrá atender al público (clientes y no clientes).
-El ***Administrador***, además de atender al público, podrá gestionar clientes, productos y otros empleados (*NO otros administradores*). Por gestión se entiende alta/baja/modificación.
-El ***Jefe***, además de atender al público y gestionar clientes, productos y otros empleados (*incluidos administradores*), puede agregar un nuevo dueño, alta y baja de administradores, y revisar ***todos los tickets*** *que se van generando por compra*.

Desarrollo:

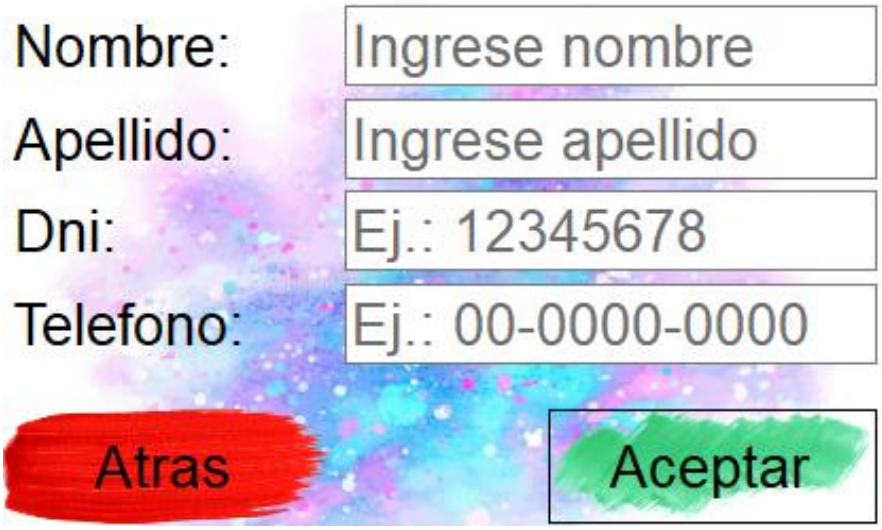
La empresa de diseño gráfico ***e-grafic*** que se encarga de proveer **Diseños** e **Impresiones** nos solicitó un sistema de atención al público, que permita atender tanto a **clientes registrados** como **clientes eventuales**.
La ventaja de aquellos clientes que ***decidan registrarse***, es que, *si realizan 7 o más compras en los últimos 30 días, entraran en la categoría de* ***Cliente Recurrente****, obteniendo asi un 5% de descuento en sus futuras compras.* Este descuento se mantendrá activo mientras el cliente siga siendo ***Cliente Recurrente***.



Además, los clientes registrados pueden gozar de otro *descuento del 5% si, en los últimos 90 días han invertido en el local un total de $25.000, entrando a la categoría de* ***Cliente VIP****.* Este descuento se mantendrá activo mientras el cliente siga siendo ***Cliente VIP.***
Ambos descuentos pueden acumularse para un total de 10% de descuento en cada compra, siempre y cuando el cliente mantenga ambas categorías.



El sistema permite dar de alta clientes, con ***nombre completo, dni y teléfono***.



De todos estos datos, solo puede ***variar*** el número de teléfono, por lo que el sistema permite modificar dicho campo.

Por otra parte, el sistema permite dar de baja clientes que, por alguna razón, deban borrarse. Si bien los ***clientes de baja*** serán borrados de los archivos, en la base de datos los vamos a mantener “ocultos”, con el estado “***No_Activo***”.

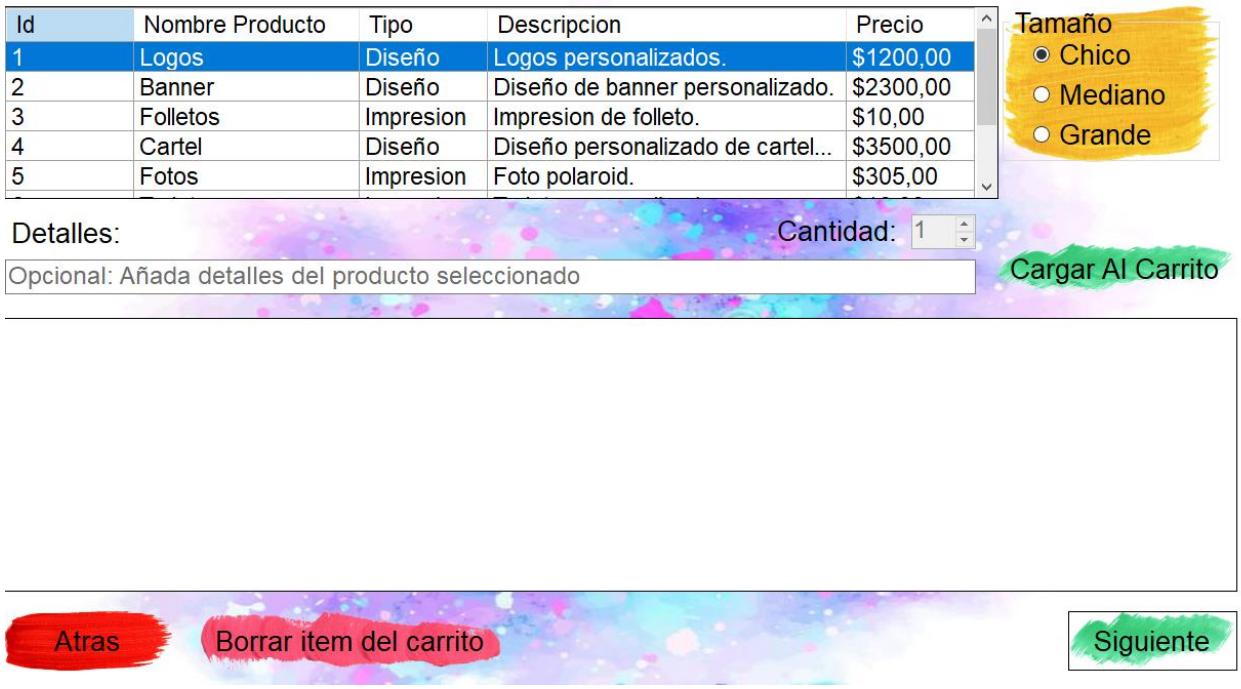
	Dni	Nombre	Apellido	Telefono	Activo
1	16987656	Roberto	Carlos	11-2222-3333	Activo
2	21659989	Juan	Martin	11-9876-7854	Activo
3	33259797	Daiana	Villar	15-1236-1198	Activo
4	34659797	Lucia	Arruti	15-1234-9876	No_Activo
5	35498796	Gonzalo	Stregaro	15-3215-9876	Activo

En la Base de datos basta con cambiar el estado de **Activo** a **No_Activo**, para que ese cliente no se cargue más en el sistema, pero la información del cliente no se pierde. (Lo mismo sucederá con los empleados). Solo el jefe podría reactivar un cliente (o empleado), a través de su DNI.

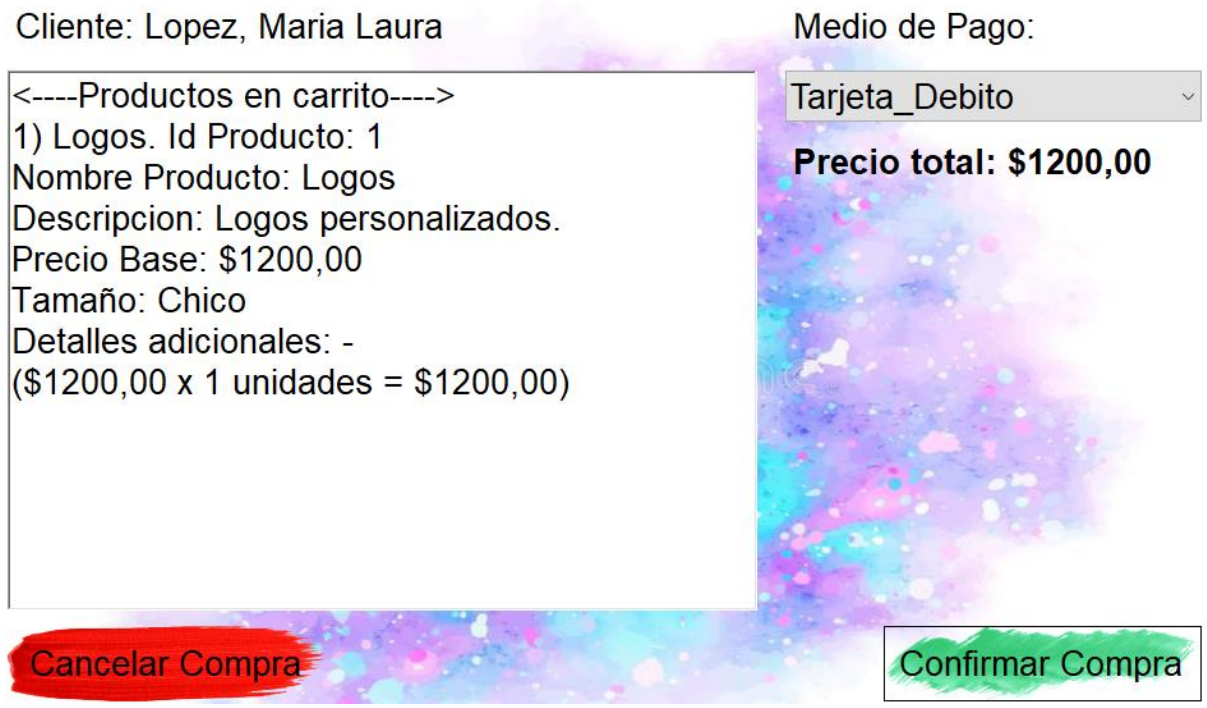
El sistema de atención permite al empleado usuario, elegir cualquiera de los productos cargados en el sistema, y ***añadirlos al carrito de compras del cliente***.
En el caso de **Impresiones**, el sistema permite establecer la cantidad de unidades que el cliente llevara, y si es ***Sin color o Con color (el color duplica el precio del producto).***
Por otro lado, los **Diseños** se venden ***por unidad únicamente***, ya que se está vendiendo el diseño de un logo, un banner o un cartel, por ejemplo. Los diseños vienen en 3 tamaños: ***Chico, mediano (duplica el costo) y grande (triplica el costo).*** Los diseños siempre vienen con color.

Adicionalmente, se permite agregar detalles (opcionales) acerca del producto que el cliente llevara, por ejemplo: “El cliente solicito que el logo se realice con colores claros”, “El cliente solicito que el banner este con colores vibrantes”.

En caso de ser necesario, el empleado puede remover ítems del carrito.

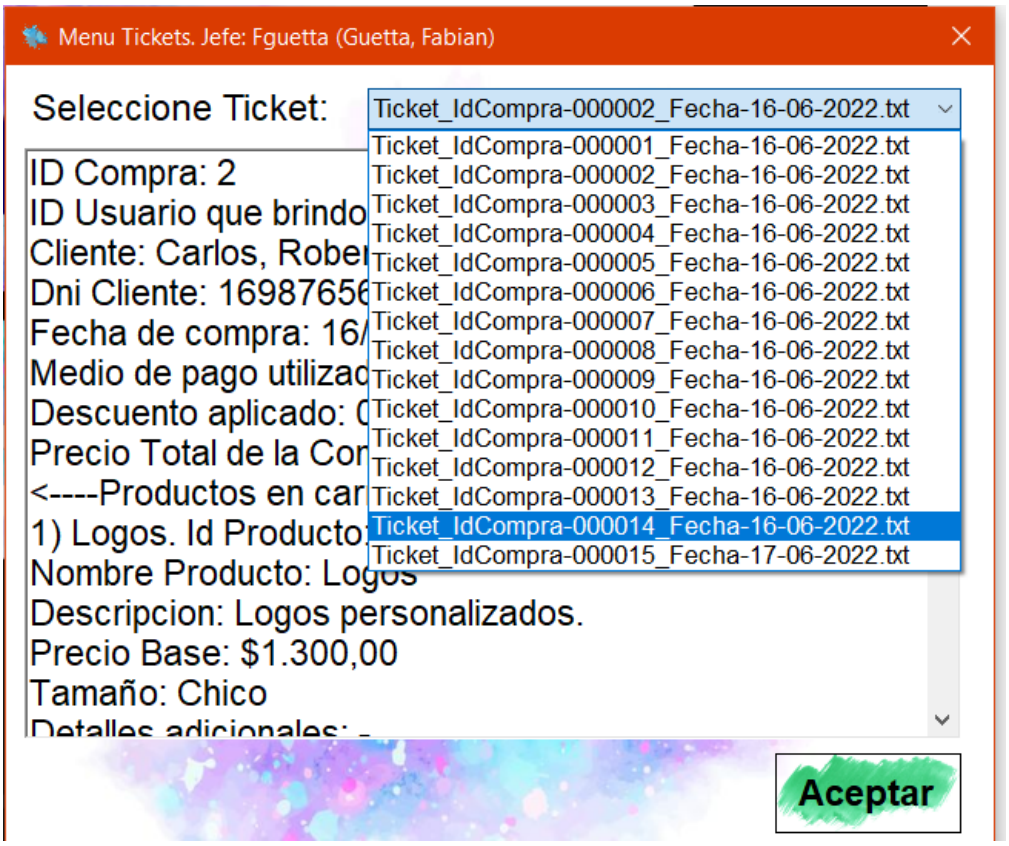


Una vez terminada la atención, se abrirá una nueva ventana de confirmación de compra, con toda la información de los productos en el carrito, así como los posibles descuentos que le corresponden a un cliente registrado y el medio de pago a utilizar.



Al confirmar la compra, saldrá un mensaje informando que la compra fue exitosa, y ***se generara un archivo .txt con la información del ticket.***

Por último, el jefe de la empresa tiene la posibilidad de acceder a todos los tickets que se hayan generado:



Al sistema de atención solicitado, se **sumó** la posibilidad de **agregar, modificar y borrar** empleados, existiendo 3 categorías: Jefe, Administrador y Empleado. También se pueden **agregar, modificar y eliminar** productos correspondientes al área de Diseño e Impresiones.

Temas implementados

Temas implementados de la clase 10 – 15: Excepciones, Genéricos, Interfaces, Archivos y Serializacion. Se deja una copia del PDF del TP3 en la solución, con el detalle de implementación de los temas mencionados.

Temas implementados de la clase 16 – 21:

Clase 16 y 17 – SQL y Conexión a Base de datos:

La clase “InteraccionConBaseDeDatos” será la encargada de manejar todo tipo de consulta SQL a la Base de datos, sea de tipo Select, Insert, Update o Delete.
Los métodos ***fundamentales*** de dicha clase son
“**EjecutarConsultaDeLecturaDeRegistros()**” y
“**EjecutarConsultaDeModificacionDeRegistros()**”.

```
/// <summary> Ejecuta una Consulta de tipo 'SELECT' para Leer Registros de la Ba ...
5 referencias
public string EjecutarConsultaDeLecturaDeRegistros(string consultaSql, LeerSqlData metodoLecturaSql)
{
    return this.EjecutarConsultaDeLecturaDeRegistros(consultaSql, metodoLecturaSql, false);
}

/// <summary> Ejecuta una Consulta de tipo 'SELECT-parametrizada' para Leer Regi ...
7 referencias
public string EjecutarConsultaDeLecturaDeRegistros(string consultaSql, LeerSqlData metodoLecturaSql, CargarParametrosSQL
{
    this.MetodoCargaDeParametrosSql = metodoCargaDeParametrosSql;

    return this.EjecutarConsultaDeLecturaDeRegistros(consultaSql, metodoLecturaSql, true);
}

/// <summary> Ejecuta una Consulta de tipo 'INSERT-UPDATE-DELETE' para modificar ...
11 referencias
public string EjecutarConsultaDeModificacionDeRegistros(string consultaSql, CargarParametrosSQL metodoCargaDeParametrosS
```

Estos métodos son utilizados por las clases ***ClienteDao***, ***EmpleadoDao***, ***ProductoDao*** y ***CompraDao***, para manipular las tablas existentes en la Base De Datos.

La Clase ***ProductoDao***, por ejemplo, utiliza los métodos para Seleccionar, Actualizar, Eliminar e Insertar de la siguiente forma:

```
public List<Producto> LeerTodosLosRegistrosDeBaseDeDatos()
{
    try
    {
        this.productos.Clear();

        string consultaSQL = "SELECT * FROM Productos";

        this.ResultadoDeLecturaDeBBDD = this.interaccionConBaseDeDatos.EjecutarConsultaDeLecturaDeRegistros(consultaSQL,

        return this.productos;
    }
    catch (Exception)
    {
        throw;
    }
}
```

```
2 referencias
public string ActualizarEnBaseDeDatos(Producto producto)
{
    try
    {
        if (producto is not null)
        {
            this.producto = producto;

            string consultaSQL = "UPDATE Productos SET Descripcion = @descripcion, Precio = @precio WHERE Id_Producto = (

            return this.interaccionConBaseDeDatos.EjecutarConsultaDeModificacionDeRegistros(consultaSQL, ((IBaseDeDatos<
        }
        throw new ArgumentNullException("Producto es NULL");
    }
    catch (Exception)
    {
        throw;
    }
}
public string EliminarDeBaseDeDatos(Producto producto)
{
    try
    {
        if (producto is not null)
        {
            this.producto = producto;

            string consulta = "DELETE FROM Productos WHERE Id_Producto = @idProducto";

            return this.interaccionConBaseDeDatos.EjecutarConsultaDeModificacionDeRegistros(consulta,

        }
        throw new ArgumentNullException("Producto es NULL");
    }
    catch (Exception)
    {
        throw;
    }
}
```

```
2 referencias
public string InsertarEnBaseDeDatos(Producto producto)
{
    try
    {
        if (producto is not null)
        {
            this.producto = producto;

            string consultaSQL;

            if (this.producto is Disenio)
            {
                consultaSQL =
                    "INSERT INTO Productos (Nombre_Producto, Descripcion, Precio, Tamanio, Categoria) " +
                    "VALUES(@nombre, @descripcion, @precio, @tamanio, @categoria)";
            }
            else
            {
                consultaSQL =
                   "INSERT INTO Productos (Nombre_Producto, Descripcion, Precio, Tiene_Color, Categoria) " +
                   "VALUES(@nombre, @descripcion, @precio, @tieneColor, @categoria)";
            }

            return this.interaccionConBaseDeDatos.EjecutarConsultaDeModificacionDeRegistros(consultaSQL, ((IBaseDe
        }
```

Clase 18 – Delegados y Expresiones Lamda:

En el proyecto hay un archivo “Delegados.cs” con los delegados utilizados:

```
namespace Entidades
{
    public delegate void CargarParametrosSQL(SqlCommand comando);

    public delegate bool LeerSqlData(SqlDataReader sqlLector);

    public delegate void CargarProgresoDeLecturaHandler(int porcentajeDeProgreso);

    public delegate void LecturaDeDatosHandler(string lecturaDeDatos);
}
```

Los delegados “**CargarParametrosSQL(SqlCommand)**” y “**LeerSqlData(SqlDataReader)**” son utilizados por la clase “**InteraccionConBaseDeDatos**”, para poder recibir ***Metodos específicos*** por parámetro, los cuales, respectivamente, cargarán parámetros en el SQLCommand o leerán la información traída desde un registro de la base de datos, a través del SqlDataReader.

En la clase ClienteDAO, por ejemplo, tenemos el siguiente método:

```
void IBaseDeDatos<Cliente>.CargarParametrosSQLParaInsertarRegistro(SqlCommand comando)
{
    if (comando is not null && this.cliente is not null)
    {
        comando.Parameters.AddWithValue("@dni", this.cliente.Dni);
        comando.Parameters.AddWithValue("@nombre", this.cliente.Nombre);
        comando.Parameters.AddWithValue("@apellido", this.cliente.Apellido);
        comando.Parameters.AddWithValue("@telefono", this.cliente.Telefono);
        comando.Parameters.AddWithValue("@activo", this.cliente.Activo.ToString());
    }
    else
    {
        throw new ArgumentNullException("Comando o cliente es NULL");
    }
}
```

Este método es VOID y recibe por parametro un SqlCommand, por lo cual **cumple** con la firma del delegado "**CargarParametrosSQL**". Este método puede enviarse como argumento al método "EjecutarConsultaDeModificacionDeRegistros()" que espera por parámetro un delegado del tipo "**CargarParametrosSQL**". De esta forma se DELEGA la tarea de cargar los parámetros en el SQLCommand, lo cual nos permite reutilizar toda la estructura de código que conecta con BBDD y hace la consulta.

Luego, desde la clase ***InteraccionConBaseDeDatos*** se utiliza el delegado:

```
13 referencias
public string EjecutarConsultaDeModificacionDeRegistros(string consultaSql, CargarParametrosSQL metodoCargaDeParametrosSql)
{
    this.filasAfectadas = 0;

    try
    {
        if (!string.IsNullOrWhiteSpace(consultaSql) && metodoCargaDeParametrosSql is not null)
        {
            this.conexion.Open();
            this.comando.CommandText = consultaSql;

            metodoCargaDeParametrosSql(this.comando);

            this.filasAfectadas = this.comando.ExecuteNonQuery();

            return $"Se han modificado {this.filasAfectadas} filas";
        }
```

Otra implementación: El método **"LeerSqlData()"** retorna Bool, y recibe un SqlDataReader, por lo que cumple con la firma del delegado **"LeerSqlData"**, debido a esto, también podrá ser enviado como argumento al método que lo requiera, para leer clientes en este caso particular.

```
bool IBaseDeDatos<Cliente>.LeerSqlData(SqlDataReader sqlLector)
{
    Cliente cliente;

    try
    {
        if (sqlLector is not null)
        {
            if (int.TryParse(sqlLector["Dni"].ToString(), out int dni) &&
                Enum.TryParse(sqlLector["Activo"].ToString(), out Persona.EstaActivo estaActivo))
            {
                string nombre = sqlLector["Nombre"].ToString();
                string apellido = sqlLector["Apellido"].ToString();
                string telefono = sqlLector["Telefono"].ToString();

                cliente = new Cliente(nombre, apellido, dni, telefono, estaActivo);
```

Luego, desde la clase ***InteraccionConBaseDeDatos***, se utiliza el delegado recibido como argumento:

```
2 referencias
private string EjecutarConsultaDeLecturaDeRegistros(string consultaSql, LeerSqlData metodoLecturaSql, bool tieneParametros)
{
    this.filasTotales = 0;
    this.filasLeidasConExito = 0;
    this.filasConError = 0;

    try
    {
        if (!string.IsNullOrWhiteSpace(consultaSql) && metodoLecturaSql is not null)
        {
            this.conexion.Open();

            this.comando.CommandText = consultaSql;

            if (tieneParametros && this.MetodoCargaDeParametrosSql is not null)...

            using (SqlDataReader sqlLector = this.comando.ExecuteReader())
            {
                while (sqlLector.Read())
                {
                    this.filasTotales++;

                    try
                    {
                        metodoLecturaSql(sqlLector);

                        this.filasLeidasConExito++;
                    }
```

También se utilizan delegados del sistema (**Action y Action<T>**) para modificar controles del windowForm desde un **hilo secundario**. Ejemplos en el ***FrmPantallaInicial***:

```
private void CargarProgresoDeDescargaEnProgressBar(int progresoDeDescarga)
{
    if (this.pBarDescargaDatos.InvokeRequired)
    {
        this.pBarDescargaDatos.Invoke(new Action<int>(this.CargarProgresoDeDescargaEnProgressBar), progresoDeDescarga);
    }
    else
    {
        if ((this.pBarDescargaDatos.Value + progresoDeDescarga) <= 100)
        {
            this.pBarDescargaDatos.Value += progresoDeDescarga;
        }
    }
}
```

```
private void LecturaDeDatos(string mensaje)
{
    if (this.lblLecturaDeDatos.InvokeRequired)
    {
        this.lblLecturaDeDatos.Invoke(new Action<string>(LecturaDeDatos), mensaje);
    }
    else
    {
        this.lblLecturaDeDatos.Text = mensaje;
    }
}
```

```
private void ResetearProgresoDeDescargaEnProgressBar()
{
    if (this.pBarDescargaDatos.InvokeRequired)
    {
        this.pBarDescargaDatos.Invoke(new Action(this.ResetearProgresoDeDescargaEnProgressBar));
    }
    else
    {
        this.pBarDescargaDatos.Value = 0;
        this.pBarDescargaDatos.Visible = false;
        this.cargaExitosa = false;
        this.lblLecturaDeDatos.Visible = false;
    }
}
```

Los otros ***delegados (Handlers)*** son utilizados para eventos, por lo que se mencionaran más adelante.

Expresiones Lambda:

En el proyecto se utilizan muchas expresiones de este tipo, principalmente acompañando a una **Task** (que se verá a continuación), pero uno de los ejemplos de implementación sin TASK, es en la clase EmpleadoDAO, donde se utiliza una

expresión que cumple con la firma del delegado “**CargarParametrosSQL**”. Esta expresión tan sencilla, se envía como argumento al método **“EjecutarConsultaDeLecturaDeRegistros**”, para que cargue los parámetros en el SqlCommand:

```
public Empleado LeerUnRegistroDeBaseDeDatosPorDni(int dni)
{
    try
    {
        this.empleados.Clear();

        string consultaSQL = "SELECT * FROM Empleados WHERE Dni = @dni";

        this.ResultadoDeLecturaDeBBDD = this.interaccionConBaseDeDatos.EjecutarConsultaDeLecturaDeRegistros
            (consultaSQL, ((IBaseDeDatos<Empleado>)this).LeerSqlData,
            (comandoSQL) => comandoSQL.Parameters.AddWithValue("@dni", dni));
```

Clase 19 - Hilos:

La fuente principal de datos de este proyecto son las bases de datos, ya que son más seguras que los archivos. Por esto, toda alta, baja o modificación se realizan sobre las bases de datos. Sin embargo, como “secundario” se respalda la información sobre archivos. Este proceso de ***guardar en archivos***, se deja corriendo en paralelo, mientras el programa sigue ejecutando.

Cada vez que se realice un alta, baja o modificación en las clases de Administrador, Cliente, o Producto, y se haya guardado en la base de datos, de forma paralela se disparara un nuevo hilo que se encargara de guardar esa modificación en su respectivo archivo.

Por ejemplo, en la clase ***Cliente***, después de haber modificado un cliente en la base de datos de forma exitosa, disparamos un nuevo hilo en paralelo que lo respalde también en un archivo, mientras tanto el hilo principal sale del método retornando True:

```
1 referencia
public static bool CargarUnClienteModificadoAlSistema(Cliente cliente)
{
    try
    {
        if (cliente is not null)
        {
            Cliente.clienteDao.ActualizarEnBaseDeDatos(cliente);

            Task.Run(() =>
            {
                try
                {
                    ((IArchivo)cliente).GuardarArchivo();
                }
                catch (Exception)
                {

                }
            });

            return true;
```

Si bien se podría utilizar la Task ***antes*** que "ActualizarEnBaseDeDatos()", la idea es que los archivos (fuente secundaria de datos) tengan el **mismo contenido** que la base de datos, pero **NO MAS**, es decir, si ocurre una excepción al intentar guardar en la Base de datos, no se desea que el archivo se guarde.

Otro ejemplo de utilización de **TASK**, lo encontramos en el Formulario **FrmGestionProductos**. Cada vez que se debe actualizar el DataGrid (alta, baja o modificación), una task se encarga de vaciar una la lista y volverla a cargar de datos, mientras el hilo principal continua sus tareas cargando el datagrid:

```
4 referencias
private void RefrescardataGrid()
{
    this.dgvListaProductos.DataSource = null;

    this.taskCargarListaProductos = Task.Run(()=> this.CargarListaProductos());

    this.CargarDataGrid();
}
```

Sin embargo, después de que el hilo principal termina de crear las columnas del datagrid, asociar las propiedades en él, y darles un estilo a las columnas, preguntara si la Task termino su tarea, ya que el hilo principal requiere de dicha lista. En caso de estar ejecutándose la Task, la esperara.

```
1 referencia
private void CargarDataGrid()
{
    this.CrearColumnasDataGrid();
    this.AsociarPropiedadesAlDataGrid();
    this.DarEstiloAColumnasDataGrid();

    if(this.taskCargarListaProductos is not null &&
       this.taskCargarListaProductos.Status == TaskStatus.Running)
    {
        this.taskCargarListaProductos.Wait();
    }

    this.dgvListaProductos.DataSource = this.productos;
    this.OrdenarColumnasDataGrid();
}
```

Otro ejemplo de ***Task***, se encuentra en el ***FrmConfirmarCompra***, en donde una **Task** verificara si el cliente es recurrente, mientras que otra **Task** verificara si el cliente es VIP. Para saber si un cliente es recurrente, primero se obtienen todas las compras realizadas por el cliente, luego se revisa una por una de las compras, buscando su fecha. Si la compra se hizo en los últimos 30 días, se incrementa un acumulador en una unidad. Finalmente, si el acumulador es mayor o igual a 7, es cliente recurrente. Por otro lado, para saber si un cliente es VIP se obtienen nuevamente todas las compras que pertenecen al cliente, luego se evalúa que la fecha de las mismas sea de los últimos 90 días, finalmente, se acumulan los gastos de cada compra, y si superan los $25.000, será un cliente VIP.
Con el incremento de compras, ***estas tareas pueden demorarse más***, por lo que se utilizan **2 task** paralelas para optimizar el tiempo de búsqueda. Finalmente, se espera que ambas hayan terminado para continuar.

```
this.rTxtCarrito.Text = this.carrito.ToString();

sb.Clear();

Task taskVerificarSiElClienteEsRecurrente = Task.Run(() =>
{
    if (this.cliente.EsClienteRecurrente)
    {
        sb.AppendLine("Hay un descuento del 5% por ser cliente recurrente!");
    }
});

Task taskVerificarSiEsClienteVIP = Task.Run(() =>
{
    if (this.cliente.EsClienteVIP)
    {
        sb.AppendLine("Hay un descuento del 5% por ser cliente VIP!");
    }
});

Task.WaitAll(taskVerificarSiElClienteEsRecurrente, taskVerificarSiEsClienteVIP);
```

Una implementación más ***notoria visualmente*** del trabajo de ***Task***, es en el ***FrmPantallaInicial***, ya que se utiliza Task para cambiarle el color al "***lblImportar***" cada 350 milisegundos (mientras no se haya cancelado la ***Task***), dando un efecto de resaltado:

```
private void FrmPantallaInicial_Load(object sender, EventArgs e)
{
    Administrador.OnCargaCompleta += this.CargarProgresoDeDescargaEnProgressBar;
    Producto.OnCargaCompleta += this.CargarProgresoDeDescargaEnProgressBar;
    Compra.OnCargaCompleta += this.CargarProgresoDeDescargaEnProgressBar;
    Cliente.OnCargaCompleta += this.CargarProgresoDeDescargaEnProgressBar;

    Administrador.OnLecturaDeFuenteDeDatos += LecturaDeDatos;
    Compra.OnLecturaDeFuenteDeDatos += LecturaDeDatos;
    Cliente.OnLecturaDeFuenteDeDatos += LecturaDeDatos;
    Producto.OnLecturaDeFuenteDeDatos += LecturaDeDatos;

    this.PrepararTemporizadorDeDescarga();

    Task.Run(() =>
    {
        while (!this.token.IsCancellationRequested)
        {
            this.ResaltarLabelImportar();
            Thread.Sleep(350);
        }
    });
}
```

```
2 referencias
private void ResaltarLabelImportar()
{
    if (this.lblImportar.InvokeRequired)
    {
        this.lblImportar.Invoke(new Action(ResaltarLabelImportar));
    }
    else
    {
        this.lblImportar.ForeColor = this.lblImportar.ForeColor == Color.Black ? Color.Red : Color.Black;
    }
}
```

Lo mismo se implementa en el ***FrmConfirmarCompra***, resaltando el "***lblTotal***", el cual será "resaltado" mientras la ***Task*** no sea cancelada.

```
    this.tokenDeCancelacion = new CancellationTokenSource();
    this.token = tokenDeCancelacion.Token;

    Task.Run(() =>
    {
        while(!token.IsCancellationRequested)
        {
            this.AlternarColorDelLblTotal();
            Thread.Sleep(350);
        }
    });
}

/// <summary> Alterna el color de las letras del 'lblTotal', entre rojo y negro.
2 referencias
private void AlternarColorDelLblTotal()
{
    if(this.lblTotal.InvokeRequired)
    {
        this.lblTotal.Invoke(new Action(this.AlternarColorDelLblTotal));
    }
    else
    {
        this.lblTotal.ForeColor = this.lblTotal.ForeColor == Color.Red ? Color.Black : Color.Red;
    }
}
```

Clase 20 – Eventos:

En el proyecto hay 2 delegados para eventos (archivo Delegados.cs):

```
    public delegate void CargarProgresoDeLecturaHandler(int porcentajeDeProgreso);

    public delegate void LecturaDeDatosHandler(string lecturaDeDatos);
}
```

Estos eventos se utilizan en las clases Administrador, Cliente, Compra y Producto. Cada vez que se ***importa información al sistema***, se cargan 4 tipos de datos, los referidos a empleados, clientes, compras y productos. Cada tipo de dato representa entonces, un 25% del total, y entre los 4 conforman el 100% de información vital del sistema.

En las clases mencionadas tenemos estos eventos:

```
97 referencias
public class Administrador : Empleado, IArchivo
{
    public static event CargarProgresoDeLecturaHandler OnCargaCompleta;
    public static event LecturaDeDatosHandler OnLecturaDeFuenteDeDatos;
```

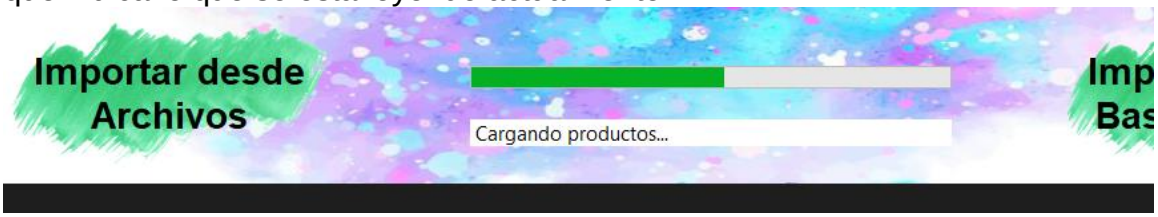
El evento **OnLecturaDeFuenteDeDatos** se disparará en su respectiva clase, cuando se esté por iniciarse la lectura de datos desde la Base de datos o desde archivos. Este evento, disparado por una Task paralela, informa en un Label del **FrmPantallaInicial** que se están “Cargando x datos…”
Una vez que se termina completamente la carga de empleados, se dispara el evento **OnCargaCompleta,** el cual informa que ya se ha cargado su correspondiente 25%.
Por ejemplo, en la clase Administrador, se encuentra el método **“CargarEmpleadosAlSistemaDesdeBBDD()”**, el cual informa que esta “Cargando empleados…”, y al finalizar la carga, informa que ha cargado su 25%.

```
public static void CargarEmpleadosAlSistemaDesdeBBDD()
{
    try
    {
        Task.Run(() =>
        {
            if (Administrador.OnLecturaDeFuenteDeDatos is not null)
            {
                Administrador.OnLecturaDeFuenteDeDatos("Cargando empleados...");
            }
        });

        List<Empleado> empleados = Administrador.empleadoDao.LeerTodosLosRegistro

        for (int i = 0; i < empleados.Count; i++)...

        if(!Administrador.SistemaPoseeJefe())...

        Task.Run(() =>
        {
            if (Administrador.OnCargaCompleta is not null)
            {
                Administrador.OnCargaCompleta(25);
            }
        });
```

Estos eventos reciben manejadores en el FrmPantallaInicial, en el evento Load:

```
private void FrmPantallaInicial_Load(object sender, EventArgs e)
{
    Administrador.OnCargaCompleta += this.CargarProgresoDeDescargaEnProgressBar;
    Producto.OnCargaCompleta += this.CargarProgresoDeDescargaEnProgressBar;
    Compra.OnCargaCompleta += this.CargarProgresoDeDescargaEnProgressBar;
    Cliente.OnCargaCompleta += this.CargarProgresoDeDescargaEnProgressBar;

    Administrador.OnLecturaDeFuenteDeDatos += LecturaDeDatos;
    Compra.OnLecturaDeFuenteDeDatos += LecturaDeDatos;
    Cliente.OnLecturaDeFuenteDeDatos += LecturaDeDatos;
    Producto.OnLecturaDeFuenteDeDatos += LecturaDeDatos;
```

El resultado: Una barra de progreso que indica lo que se ha cargado, y un label que indica lo que se está leyendo actualmente.



Una vez que se haya cargado toda la información con éxito, y el valor de la barra de progreso haya llegado al 100%, se permitirá el inicio de sesión. Para lograr esto, se utiliza un ***Timer***, el cual estará revisando cada 750 milisegundos si la carga se completó. El ***Timer***, en su evento ***Tick***, tiene suscrito un método que define tal comportamiento:

```
1 referencia
private void PrepararTemporizadorDeDescarga()
{
    this.tmrEvaluarDescarga.Tick += this.VerificarAvanceDeProgressBar;
    this.tmrEvaluarDescarga.Interval = 750;
}

/// <summary> Verifica el avance de la progressBar
2 referencias
private void VerificarAvanceDeProgressBar(object sender, EventArgs e)
{
    if (this.pBarDescargaDatos.Value == 100 && this.cargaExitosa)
    {
        this.LimpiarEventosYTrabajosParalelos();

        this.Hide();

        FrmLogin login = new FrmLogin();
        login.ShowDialog();

        this.Close();
    }
```

Clase 21 – Métodos de extensión:

Hay una clase específica, ***ExtensionString***, donde hay métodos de extensión, por ejemplo:

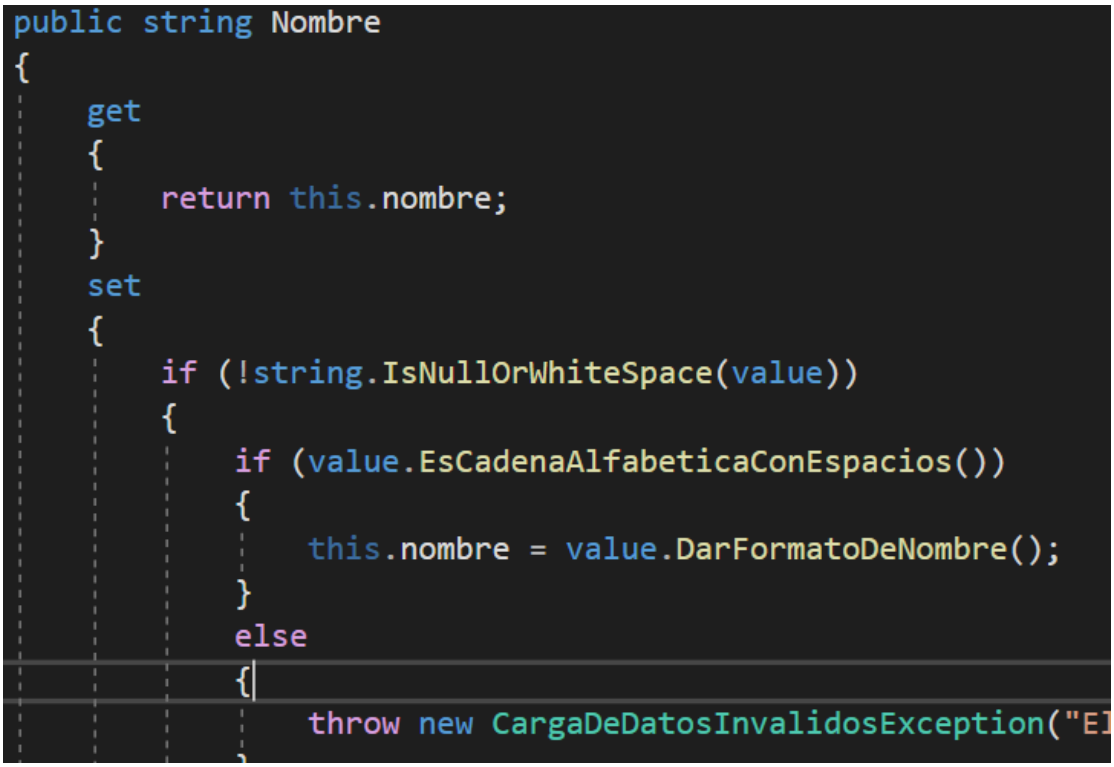
```
/// <summary>
/// Evalua si una cadena respeta el formato telefonico.
/// </summary>
/// <param name="cadena">Cadena que se evaluara.</param>
/// <returns>True si la cadena respeta el formato telefonico (00-0000-0000), caso contrario, False.</returns>
3 referencias | 2/2 pasando
public static bool EsFormatoTelefonico(this string cadena)
{
    return ExtensionString.EsCadenaValida(cadena, "^[0-9]{2}-[0-9]{4}-[0-9]{4}$");
}
```

```
/// <summary>
/// Da el formato de nombre (primera letra de cada palabra en Mayuscula, resto de la palabra en minusculas).
/// Tambien se encarga de borrar espacios EXTRAS, dejando, de ser necesario, un unico espacio en blanco entre palabr
/// </summary>
/// <param name="cadena">Cadena que se evaluara.</param>
/// <returns>Una copia de la cadena con la primera letra de cada palabra convertida a mayusculas, el resto en minusc
5 referencias | 1/1 pasando
public static string DarFormatoDeNombre(this string cadena)
{
    string cadenaAuxiliar = string.Empty;

    if (!string.IsNullOrWhiteSpace(cadena))
    {
        cadenaAuxiliar = ExtensionString.ConvertirCaracterInicialDeCadaPalabraAMayuscula(cadena);
    }

    return cadenaAuxiliar;
}
```

Implementación en clase Cliente (entre otras clases que también lo implementan):

```
5 referencias
public string Telefono
{
    get
    {
        return this.telefono;
    }
    set
    {
        if (!string.IsNullOrWhiteSpace(value))
        {
            if(value.EsFormatoTelefonico())
            {
                this.telefono = value;
            }
            else
            {
                throw new CargaDeDatosInvalidosException("Te
            }
        }
```

La clase Persona, en su propiedad Nombre implementa 2 métodos de extensión: ***EsCadenaAlfabeticaConEspacios***() y ***DarFormatoDeNombre***()



```
public string Nombre
{
    get
    {
        return this.nombre;
    }
    set
    {
        if (!string.IsNullOrWhiteSpace(value))
        {
            if (value.EsCadenaAlfabeticaConEspacios())
            {
                this.nombre = value.DarFormatoDeNombre();
            }
            else
            {
                throw new CargaDeDatosInvalidosException("El
```