Universidad Nacional de Ingeniería

Área de Conocimiento de Tecnología de la Información y Comunicación

Ingeniería de Sistemas

Programación II

Mejora del Proyecto EntidadFinanciera2m6

Integrantes:

- Hillary Daned Ordoñez Diaz
- Ismaurily Tatiana Pichardo Rizo
- Aidan Adrian Kelly Ortiz
- Erick Mateo Sequeira Centeno

Grupo: 2M6-SIS

Docente: Abel Edmundo Marin Reyes

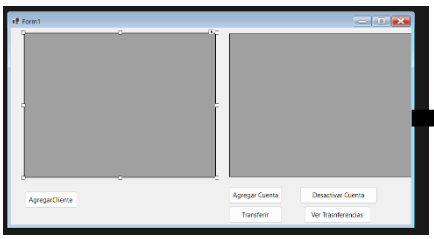
Managua, mayo del 2025

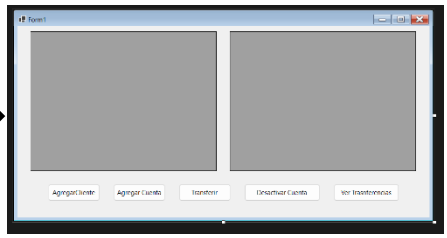
La aplicación **EntidadFinanciera2M6** fue objeto de un proceso integral de revisión y refactorización con el propósito de aplicar buenas prácticas de programación orientada a objetos y principios de arquitectura limpia. Se realizaron modificaciones en diversos formularios, clases y componentes de la solución, enfocándose en mejorar la legibilidad, mantenibilidad, reutilización del código y separación de responsabilidades. A continuación, se detallan los principales ajustes organizados por módulo, explicando su propósito, ejecución y beneficios dentro del sistema.

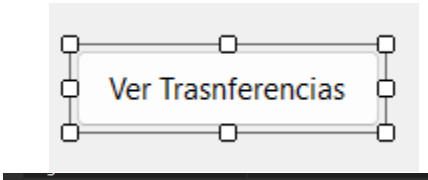
I. Form1

Reestructuración de la interfaz de usuario y nombres de controles

Se reorganizó la disposición de los botones y elementos gráficos para proporcionar una interfaz más clara e intuitiva. Además, se renombraron varios controles, como el botón Ver transferencias de su nombre genérico "button1", que ahora se llama **btnTransferencias**, reflejando su propósito real. Esto mejora la legibilidad del código y facilita su mantenimiento al eliminar ambigüedades.

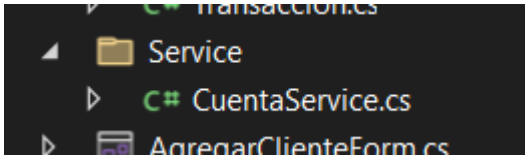






Design	
(Name)	btnTransferencias
GenerateMember	True
Locked	False
Modifiers	Private

Separación de la lógica de negocio



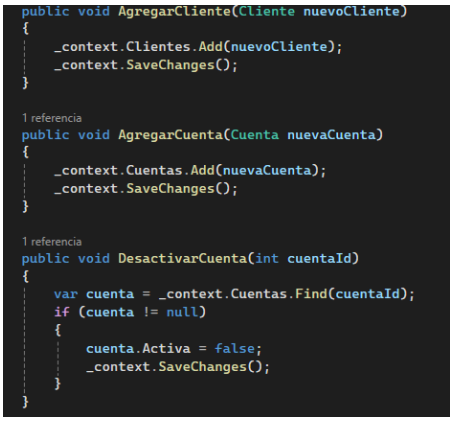
La lógica de negocio que anteriormente residía en el formulario fue trasladada a una nueva clase denominada **CuentaService.cs**. Esta clase contiene métodos como ObtenerCuentasPorId, ObtenerClientesConCuenta, AgregarCuenta, entre otros, centralizando las operaciones relacionadas con clientes y cuentas. Esta separación permite mantener el formulario enfocado exclusivamente en la lógica de presentación, promoviendo el principio de responsabilidad única y facilitando la reutilización del código en otros formularios.

```
namespace EntidadFinanciera2M6.Services
{
    3 referencias
    public class CuentaService
    {
        private readonly EntidadFinancieraContext _context;

        1 referencia
        public CuentaService(EntidadFinancieraContext context)
        {
            _context = context;
        }

        1 referencia
        public List<Cliente> ObtenerClientesConCuentas()
        {
            return _context.Clientes.Include(c => c.Cuentas).ToList();
        }

        1 referencia
        public List<object> ObtenerCuentasActivas()
        {
            return _context.Cuentas
                .Include(c => c.Cliente)
                .Where(c => c.Activa)
                .Select(c => new
                {
                    c.CuentaId,
                    c.NumeroCuenta,
                    c.Saldo,
                    c.Activa,
                    c.ClienteId,
                    ClienteNombre = c.Cliente.Nombre
                }).ToList<object>();
        }
```



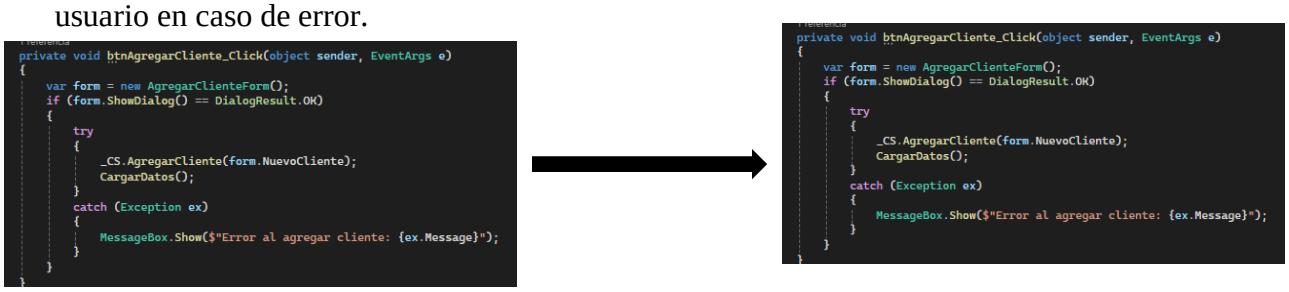
```
public void AgregarCliente(Cliente nuevoCliente)
{
    _context.Clientes.Add(nuevoCliente);
    _context.SaveChanges();
}

1 referencia
public void AgregarCuenta(Cuenta nuevaCuenta)
{
    _context.Cuentas.Add(nuevaCuenta);
    _context.SaveChanges();
}

1 referencia
public void DesactivarCuenta(int cuentaId)
{
    var cuenta = _context.Cuentas.Find(cuentaId);
    if (cuenta != null)
    {
        cuenta.Activa = false;
        _context.SaveChanges();
    }
}
```

Manejo de errores con bloques try-catch

Se introdujeron bloques try-catch en puntos críticos como las operaciones de transferencia, con el fin de manejar excepciones de manera controlada y mejorar la estabilidad general de la aplicación. Esta práctica previene cierres inesperados y proporciona mensajes claros al usuario en caso de error.



```
private void btnAgregarCliente_Click(object sender, EventArgs e)
{
    var form = new AgregarClienteForm();
    if (form.ShowDialog() == DialogResult.OK)
    {
        try
        {
            _CS.AgregarCliente(form.NuevoCliente);
            CargarDatos();
        }
        catch (Exception ex)
        {
            MessageBox.Show($"Error al agregar cliente: {ex.Message}");
        }
    }
}
```

```
private void btnAgregarCliente_Click(object sender, EventArgs e)
{
    var form = new AgregarClienteForm();
    if (form.ShowDialog() == DialogResult.OK)
    {
        try
        {
            _CS.AgregarCliente(form.NuevoCliente);
            CargarDatos();
        }
        catch (Exception ex)
        {
            MessageBox.Show($"Error al agregar cliente: {ex.Message}");
        }
    }
}
```

Establecimiento de una convención de nombres coherente

Se renombraron eventos y métodos siguiendo un patrón descriptivo y significativo. Por ejemplo, el evento button1_Click fue reemplazado por btnHistorialTransferencia_Click. Esto facilita la navegación en el código y mejora la comprensión general del flujo del programa.

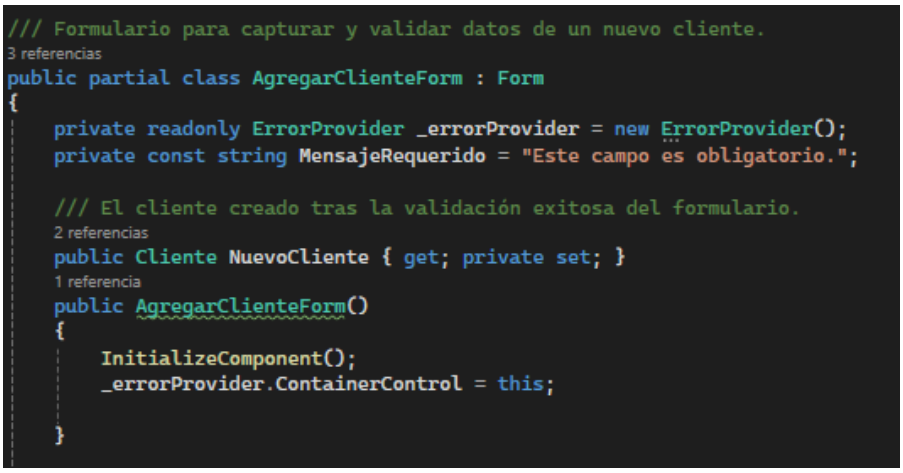
```
1 referencia
private void button1_Click(object sender, EventArgs e)
{
    var form = new Form2();
    form.ShowDialog();
}
```

```
1 referencia
private void btnHistorialTransferencias_Click(object sender, EventArgs e)
{
    var form = new Transacciones();
    form.ShowDialog();
}
```

II. AgregarClienteForm

Documentación de funciones

Se añadieron comentarios explicativos en todas las funciones del módulo, describiendo claramente su propósito, parámetros y retorno. Esto promueve la colaboración entre desarrolladores y facilita la futura expansión o depuración del sistema.



```
/// Formulario para capturar y validar datos de un nuevo cliente.
3 referencias
public partial class AgregarClienteForm : Form
{
    private readonly ErrorProvider _errorProvider = new ErrorProvider();
    private const string MensajeRequerido = "Este campo es obligatorio.";

    /// El cliente creado tras la validación exitosa del formulario.
    2 referencias
    public Cliente NuevoCliente { get; private set; }
    1 referencia
    public AgregarClienteForm()
    {
        InitializeComponent();
        _errorProvider.ContainerControl = this;

    }
```

Implementación de validación de campos con ErrorProvider

Se integró el componente **ErrorProvider** junto con una función de validación previa a la ejecución. Esta validación mejora la experiencia del usuario al señalar de manera visual los errores en la entrada de datos, previniendo el procesamiento de información incompleta o incorrecta. Esta estrategia incrementa la robustez de la aplicación y reduce los errores en tiempo de ejecución.

```
/// Valida los campos del formulario y marca errores si es necesario.
/// True si todos los campos son válidos; false en caso contrario.
0 referencias
private bool ValidarCampos()
{
    bool esValido = true;
    _errorProvider.Clear();

    // Validar Nombre
    var nombre = txtNombre.Text?.Trim();
    if (string.IsNullOrEmpty(nombre))
    {
        _errorProvider.SetError(txtNombre, MensajeRequerido);
        esValido = false;
    }

    // Validar Identificación
    var identificacion = txtIdentificacion.Text?.Trim();
    if (string.IsNullOrEmpty(identificacion))
    {
        _errorProvider.SetError(txtIdentificacion, MensajeRequerido);
        esValido = false;
    }

    return esValido;
}
```

III. AgregarCuentasForm

Corrección de nombre del formulario

Se corrigió un error tipográfico en el nombre del formulario, cambiando "AgregarCuetasForm" por **AgregarCuentasForm**, garantizando consistencia en toda la solución.

```
1 referencia
public partial class AgregarCuetasForm : Form
{
```

```
//Formulario para agregar una nueva cuenta asociada a un cliente.
3 referencias
public partial class AgregarCuentasForm : Form
{
    private readonly ErrorProvider _errorProvider = new ErrorProvider();
    private readonly int _clienteId;

    //Cuenta creada tras aceptar el formulario.
    2 referencias
    public Cuenta NuevaCuenta { get; private set; }
    private int _cienteId;
    1 referencia
    public AgregarCuentasForm(int clienteId)
    {
        InitializeComponent();
        _cienteId = clienteId;
        _errorProvider.ContainerControl = this;
    }
```

Mejora de la encapsulación y validación

El campo clienteId fue encapsulado como **_clienteId**, reforzando el principio de ocultamiento de información. Asimismo, se agregó una validación explícita para evitar la creación de cuentas sin número asignado.

```
//Cuenta creada tras aceptar el formulario.
2 referencias
public Cuenta NuevaCuenta { get; private set; }
private int _cienteId;
1 referencia
public AgregarCuetasForm(int clienteId)
{
    InitializeComponent();
    _cienteId = clienteId;
    _errorProvider.ContainerControl = this;
}
```

Organización de la creación de objetos

La creación del objeto NuevaCuenta se estructuró de forma clara y directa, asignando todas sus propiedades de manera ordenada en un solo bloque. Además, se refactorizó el método btnAceptar_Click para cumplir con el principio de responsabilidad única, delegando tareas específicas y evitando sobrecarga de lógica.

```
1 referencia
private void btnAceptar_Click(object sender, EventArgs e)
{
    if (string.IsNullOrWhiteSpace(txtNombre.Text) || string.IsNullOrWhiteSpace(txtIdentificacion.Text))
    {
        MessageBox.Show("Todos los campos son requeridos");
        return;
    }
    NuevoCliente = new Cliente
    {
        Nombre = txtNombre.Text,
        Identificacion = txtIdentificacion.Text
    };
    DialogResult = DialogResult.OK;
    Close();
```

```
/// Evento clic en "Aceptar": valida, crea el objeto Cliente y cierra el formulario.

private void btnAceptar_Click(object sender, EventArgs e)
{
    if (!ValidarCampos())
        return;

    NuevoCliente = new Cliente
    {
        Nombre = txtNombre.Text.Trim(),
        Identificacion = txtIdentificacion.Text.Trim()
    };

    DialogResult = DialogResult.OK;
    Close();
}
```

IV. TransferenciaForms

Renombramiento de métodos

El método ambiguo RealizarTransacc fue renombrado a **RealizarTransferencia**, describiendo con precisión su funcionalidad específica: gestionar transferencias entre cuentas.

```
1 referencia
private void RealizarTransacc(int origenCuenta, int destinoCuenta, decimal monto)
{
```

```
1 referencia
private void RealizarTransferencia(int origenCuenta, int destinoCuenta, decimal monto)
{
```

Ampliación y centralización de la validación

Se reemplazó la lógica superficial de validación por una más robusta contenida en **CuentaService**. El nuevo método valida que las cuentas existan, que el saldo sea suficiente y que el monto sea mayor a cero. Esto evita errores antes de realizar cambios en los datos, garantizando la integridad del sistema y permitiendo reutilizar la lógica desde múltiples formularios.

```
public partial class TransferenciaForms : Form
{
    private readonly ErrorProvider _errorProvider = new ErrorProvider();
    private readonly CuentaService _CS;
    private readonly int _cuentaOrigenId;
    private readonly int _cuentaDestinoId;
```

Corrección de errores tipográficos

Variables como _cuentaOrignId fueron renombradas a **_cuentaOrigenId**. Este tipo de mejoras evita confusión y favorece un código más limpio y comprensible.

```
1 referencia
private void CargarCuentas()
{
    var origen = _CS.ObtenerCuentaPorId(_cuentaOrigenId);
    var destino = _CS.ObtenerCuentaPorId(_cuentaDestinoId);

    lblCuentaOrigen.Text = $"Origen: {origen.Cliente.Nombre} - {origen.NumeroCuenta}";
    lblCuentaDestino.Text = $"Destino: {destino.Cliente.Nombre} - {destino.NumeroCuenta}";
    lblSaldo.Text = $"Saldo Disponible: {origen.Saldo:C}";
}
```

Encapsulamiento de propiedades

La propiedad Monto fue encapsulada y su setter modificado para rechazar valores no válidos (como cero o negativos). Esta protección evita inconsistencias y errores financieros, siguiendo buenas prácticas de validación interna.

```
// Valida el monto ingresado y marca error si no es válido.
1 referencia
private bool ValidarMonto()
{
    _errorProvider.Clear();
    if (numMonto.Value <= 0)
    {
        _errorProvider.SetError(numMonto, "Debe ingresar un monto mayor que cero.");
        return false;
    }
    return true;
}
```

V. Formularios de Transacciones

Corrección del orden de ejecución en el constructor

Se reorganizó el constructor para invocar primero InitializeComponent() y luego CargarTransacciones(). Esta corrección previene excepciones de referencia nula que ocurrían al acceder a controles no inicializados.

```
private readonly EntidadFinancieraContext ef;
1 referencia
public Transacciones()
{
    InitializeComponent();
    ef = new EntidadFinancieraContext();
    CargarTransacciones();

}
```

Encapsulamiento del DbContext

El DbContext fue declarado como **readonly** y se inicializó en el constructor. Esta mejora garantiza una única instancia para el ciclo de vida del formulario, reduciendo riesgos de errores por cambios accidentales y favoreciendo la estabilidad del acceso a datos.

```
private EntidadFinancieraContext ef = new EntidadFinancieraContext();
```

```
private readonly EntidadFinancieraContext ef;
1 referencia
```

Validación previa a la eliminación de registros

Se agregó una verificación en el evento btnEliminar_Click para comprobar que haya una fila seleccionada antes de intentar eliminar. Esta medida evita errores por accesos inválidos y mejora la experiencia de usuario. Así mismo Se implementó un mensaje de confirmación antes de eliminar registros. Esto protege contra eliminaciones accidentales, permitiendo que el usuario cancele la operación si fue ejecutada por error. Por último, este proceso fue envuelto en un bloque try-catch para capturar y gestionar adecuadamente posibles excepciones. Esto evita cierres inesperados de la aplicación y proporciona retroalimentación clara sobre fallos.

```
3 referencias
public partial class Form2 : Form
{
    private EntidadFinancieraContext ef = new EntidadFinancieraContext();
        1 referencia
        public Form2()
    {
        InitializeComponent();
        Cargar();
    }

    1 referencia
    private void Cargar()
    {
        dataGridView1.DataSource = ef.Transacciones.ToList();
    }
}
```

```
private void btnEliminar_Click(object sender, EventArgs e)
{
    if (dgvTransacciones.SelectedRows.Count == 0)
    {
        MessageBox.Show("Seleccione al menos una transacción para eliminar.", "Advertencia", MessageBoxButtons.OK, MessageBoxIcon.Warning);
        return;
    }

    var confirm = MessageBox.Show("¿Está seguro que desea eliminar las transacciones seleccionadas?", "Confirmar eliminación", MessageBoxButtons.YesNo, MessageBoxIcon.Question);
    if (confirm != DialogResult.Yes)
        return;

    try
    {
        foreach (DataGridViewRow fila in dgvTransacciones.SelectedRows)
        {
            int transaccionId = Convert.ToInt32(fila.Cells["TransaccionId"].Value);
            var transaccion = ef.Transacciones.Find(transaccionId);
            if (transaccion != null)
            {
                ef.Transacciones.Remove(transaccion);
            }
        }

        ef.SaveChanges();
        MessageBox.Show("Transacciones eliminadas correctamente.", "Éxito", MessageBoxButtons.OK, MessageBoxIcon.Information);
        CargarTransacciones();
    }
    catch (Exception ex)
    {
        MessageBox.Show($"Error al eliminar transacciones: {ex.Message}", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
```

Conclusión

La refactorización realizada representó un paso fundamental para mejorar la calidad estructural y funcional de la aplicación. A lo largo del proceso se abordaron múltiples aspectos clave: reorganización de la interfaz de usuario, renombramiento coherente de controles y métodos, encapsulamiento de propiedades, validación robusta de datos, manejo adecuado de excepciones y una clara separación entre las capas de presentación y lógica de negocio.

Aplicar buenas prácticas de programación orientada a objetos permitió mejorar significativamente la mantenibilidad, la legibilidad del código y la reutilización de componentes. La implementación de una arquitectura limpia y modular facilitó la reducción del acoplamiento entre partes del sistema, permitiendo su evolución de forma ordenada y segura.

La importancia de seguir buenas prácticas radica en que no solo mejoran la calidad técnica del código, sino que también promueven el trabajo colaborativo, previenen errores en producción, agilizan el desarrollo de nuevas funcionalidades y prolongan la vida útil de la aplicación. Un código bien estructurado es más fácil de entender, probar, depurar y escalar.

En definitiva, refactorizar con enfoque profesional y adoptar estándares de calidad en el desarrollo son elementos clave para construir aplicaciones sostenibles, robustas y preparadas para crecer y adaptarse a nuevas necesidades.