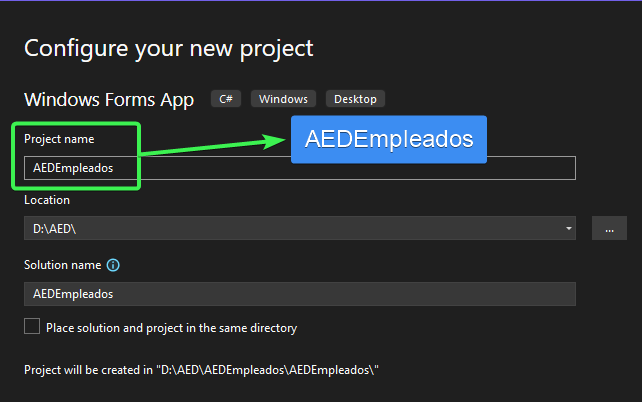
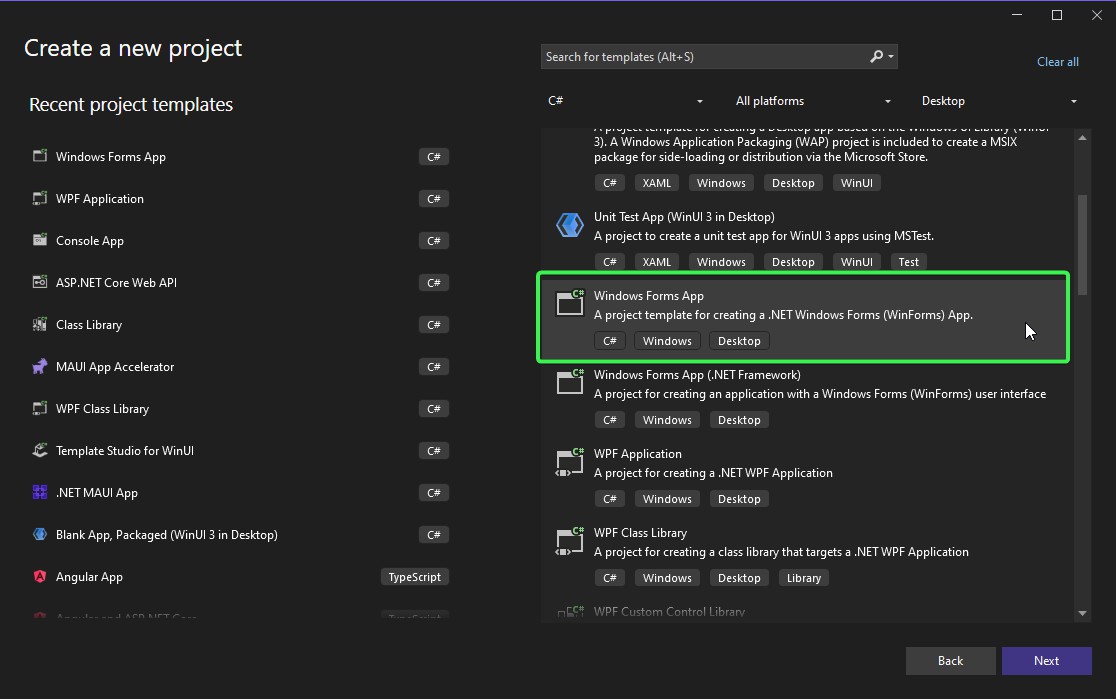
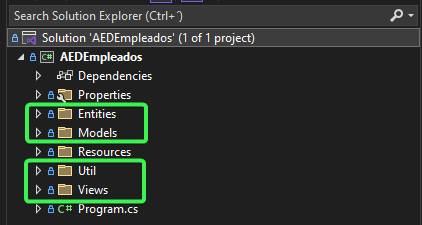
Sistema para registro y aumento de salario de los empleados

1. **Creación del proyecto en Windows Forms**



1. **Estructura del proyecto a utilizar:**



Crear cada una de las carpetas que se muestran en la imagen para una mejor organización del código a la hora de trabajar, separando la lógica de negocio de las entidades e interfaz de usuario, a continuación una breve explicación:

**Entities (Entidades):** Representa las entidades de la aplicación, es decir, los objetos que forman parte del dominio del problema. Estos son los datos con los que trabajaremos y que a su vez dan forma al cómo se realizan cada una de las operaciones [CRUD](https://en.wikipedia.org/wiki/Create%2C_read%2C_update_and_delete) dentro de la aplicación.

**Models (Modelos):** La carpeta Models es donde se maneja la lógica de negocio y las operaciones relacionadas con los datos. A diferencia de las entidades, que son solo representaciones de datos, los modelos contienen la lógica que define cómo interactuar con esas entidades.

**Util (Utilidades):** La carpeta Util contiene funciones y clases de utilidad que no pertenecen directamente a una entidad o modelo, pero que son necesarias para el funcionamiento de la aplicación. Estas clases proporcionan métodos que pueden ser reutilizados en diferentes partes del proyecto.

**Views (Vistas):** La carpeta Views contiene la interfaz de usuario de tu aplicación, en este caso, los formularios de Windows Forms. Aquí es donde los usuarios interactúan directamente con la aplicación. Las vistas son responsables de mostrar los datos y de recibir la entrada del usuario (input).

*Para más información podemos visitar****:***

* *Patrón de diseño MVC*

[Model–view–controller - Wikipedia](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller)

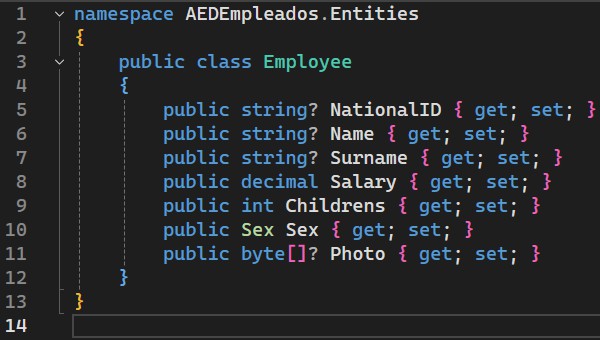
* *Diferencia entre Entity y Objeto*

[Is there a difference between an Entity and a object? - Stack Overflow](https://stackoverflow.com/questions/695934/is-there-a-difference-between-an-entity-and-a-object)

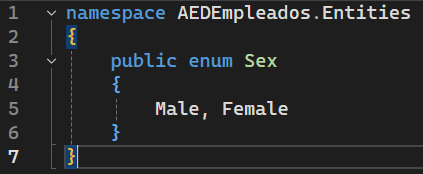
1. **Creación de las entidades de la aplicación**

Dentro de la carpeta Entities vamos a crear una clase llamada **Employee** y a su vez un enum llamado **Sex**. (Para crear enums en Visual Studio, creamos una clase llamada **Sex** y una vez generado el archivo Sex.cs reemplazamos class por enum)

Clase **Employee**

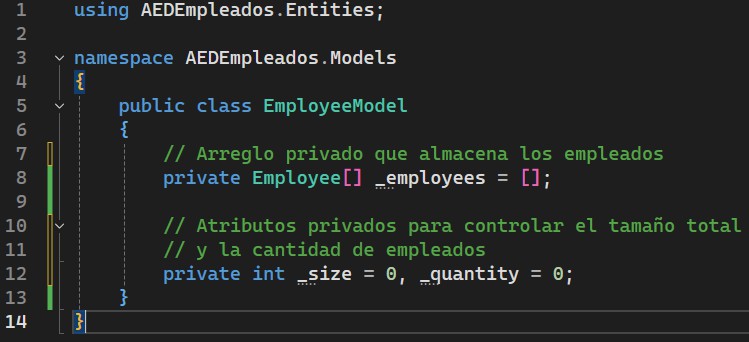


Enum **Sex**

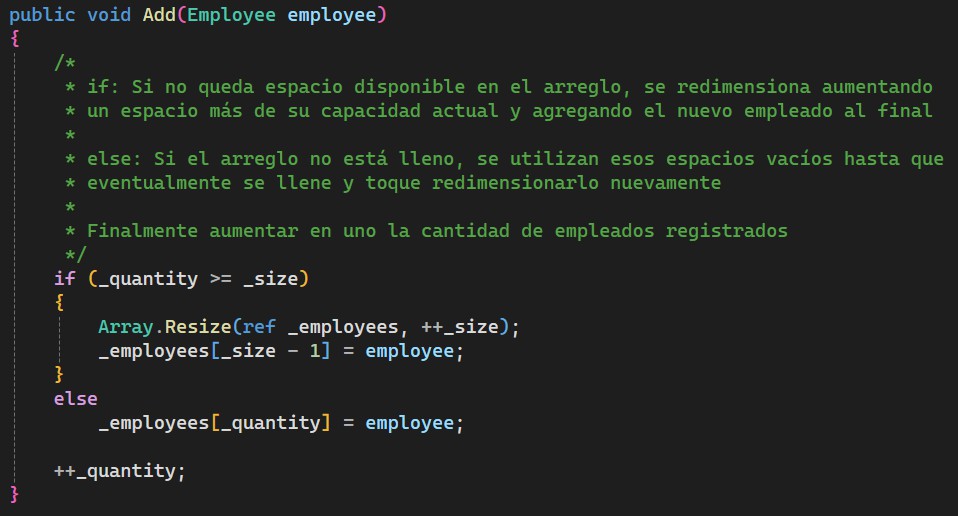


1. **Creación del modelo de datos para los empleados**

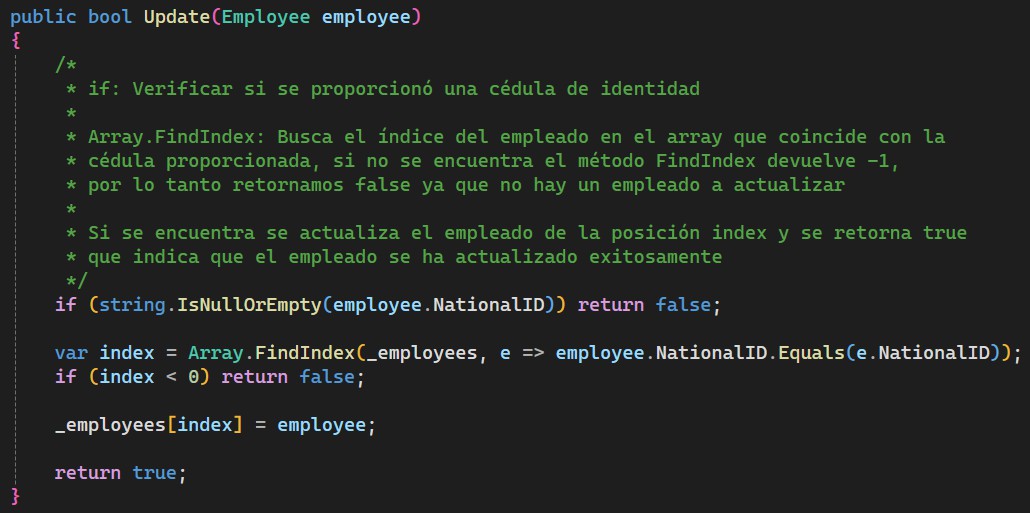
Dentro de la carpeta Models vamos a crear una clase llamada **EmployeeModel** la cual va a contener toda la lógica de acceso a los datos de nuestra aplicación y esta a su vez será compartida para su uso desde cualquier otra parte (formularios por ejemplo). Es una capa que se encarga de abstraer toda la manipulación de los datos (crear, actualizar, eliminar, buscar, realizar cálculos, etc) y solo expone las métodos para interactuar con el arreglo **(Employee[])** más no permite que se pueda acceder a él desde fuera, todo para mantener la integridad de los datos.



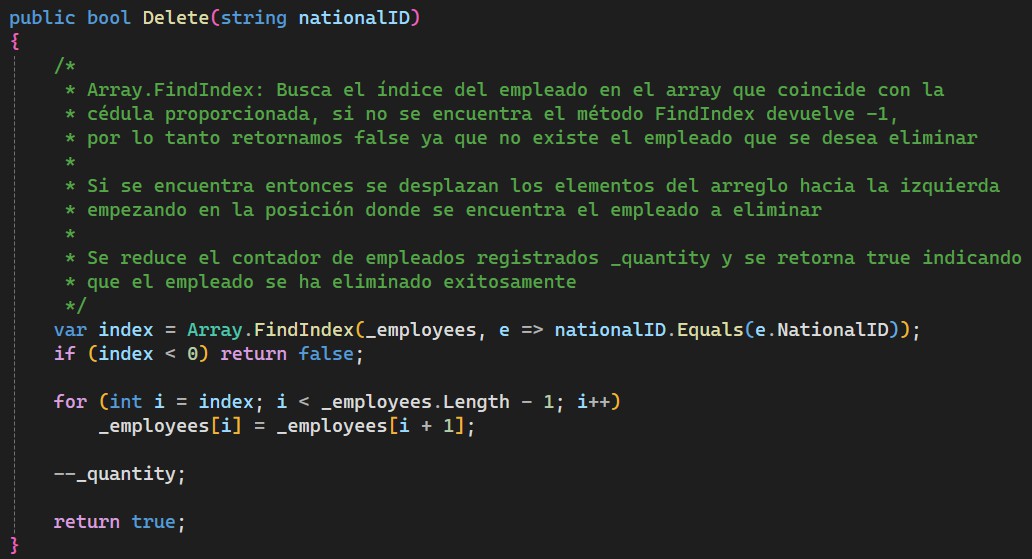
Ahora creamos un método que nos permita agregar nuevos empleados al arreglo



De esta forma solo redimensionamos el arreglo cuando es necesario, ya que en la siguiente situación donde se podrían registrar 2 empleados seguidos, se elimina uno de ellos (cualquiera) y luego se registra un tercer empleado, si no se implementara esta lógica al registrar el tercer empleado estaríamos reservando más memoria de la que necesitamos de forma innecesaria, ya que el arreglo tenía espacio disponible para almacenar ese tercer empleado sin necesidad de reservar un espacio más en memoria, ese espacio disponible sería el espacio ocupado por el empleado anterior que luego se eliminó.

Ahora implementamos el método para poder actualizar un empleado por medio de su cédula de identidad la cual tendremos que garantizar de que sea única (esa validación se realiza en otro lado)

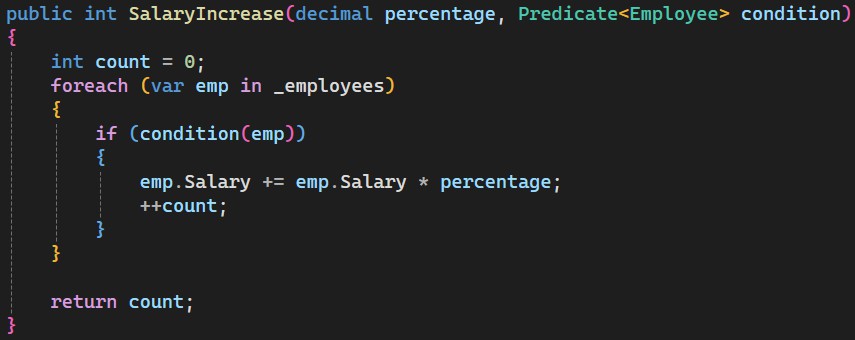
Lógica para eliminar un empleado por medio de su cédula de identidad si existe



Lógica del método GetAll para obtener todos los empleados activos, para ello devolvemos un arreglo que va desde el primer empleado hasta el que está en la posición \_quantity que sería el último empleado registrado, ignorando los datos basura que pueden haber al final del arreglo y si está vacío simplemente se ignoran esos espacios y se devuelven igualmente los registrados



Ahora agregamos un método que nos permita obtener el salario promedio de todos los empleados registrados de la siguiente forma:

Luego agregaremos un método que nos permita realizar un aumento de salario a los empleados, dado un porcentaje y una condición que nosotros deseemos:

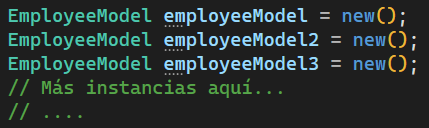
El método SalaryIncrease recibe por parámetro primeramente el porcentaje que se aplicará al aumento y un [**Predicate**](https://learn.microsoft.com/en-us/dotnet/api/system.predicate-1?view=net-8.0), delegado o condición booleana a aplicar sobre la lista, la cual se evaluará elemento a elemento y todos los que cumplan la condición se les dará el aumento con el porcentaje proporcionado. Finalmente el método retorna un entero que indica la cantidad de empleados que se vieron beneficiados por el aumento de salario.

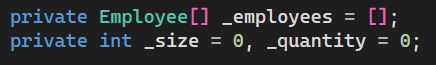
Para más información podemos visitar:

[Predicate<T> Delegate (System) | Microsoft Learn](https://learn.microsoft.com/en-us/dotnet/api/system.predicate-1?view=net-8.0) [What is a predicate in c#? - Stack Overflow](https://stackoverflow.com/questions/1710301/what-is-a-predicate-in-c)

Adicionalmente agregamos un método que nos permite buscar empleados por medio de su cédula de identidad que nos servirá para realizar la validación de cédulas ya existentes cuando registremos un nuevo empleado desde el formulario. Tomamos sólo los empleados activos (hasta **\_quantity**) y comparamos buscando uno que coincida con la cédula proporcionada).

Con esto concluimos los métodos que tendrá la clase **EmployeeModel**, pero aún queda algo pendiente y es lo siguiente…





Cada vez que creamos una instancia de la clase **EmployeeModel** estamos creando un nuevo arreglo vacío de tipo empleado y lo que queremos es que nuestro modelo de datos sea el mismo para cualquier otra clase o formulario que vaya a utilizarla sin importar cuantas instancias se puedan crear o desde donde se acceda, para ello tenemos algunas soluciones:

1. **Patrón de diseño Singleton**

Este patrón de diseño restringe la creación de instancias de la clase **EmployeeModel** a una única instancia que además será compartida por todos los que deseen usar la clase, a su vez proporciona un punto de acceso global para todos y esta sería una forma de mantener un solo arreglo de tipo **Employee[]** en toda la ejecución del programa.

Link: [Singleton pattern - Wikipedia](https://en.wikipedia.org/wiki/Singleton_pattern)

1. **Declarar el arreglo Employee[] como estático**

Al hacer esto el arreglo de empleados ahora pertenece a la clase y no a sus instancias, por lo tanto se podrán crear muchas instancias de **EmploeeModel** pero todas tendrán la misma referencia del arreglo **Employee[]** ya que es un campo de la clase.

Link: [Static Classes and Static Class Members - C# | Microsoft Learn](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members)

1. **Declara la clase EmployeeModel como estática**

Una clase estática no nos permite crear instancias de ella por lo tanto todos sus miembros o atributos le pertenecen a la clase y de esta forma también podemos garantizar de que habrá una sola instancia del arreglo **Employee[]** en toda la aplicación.

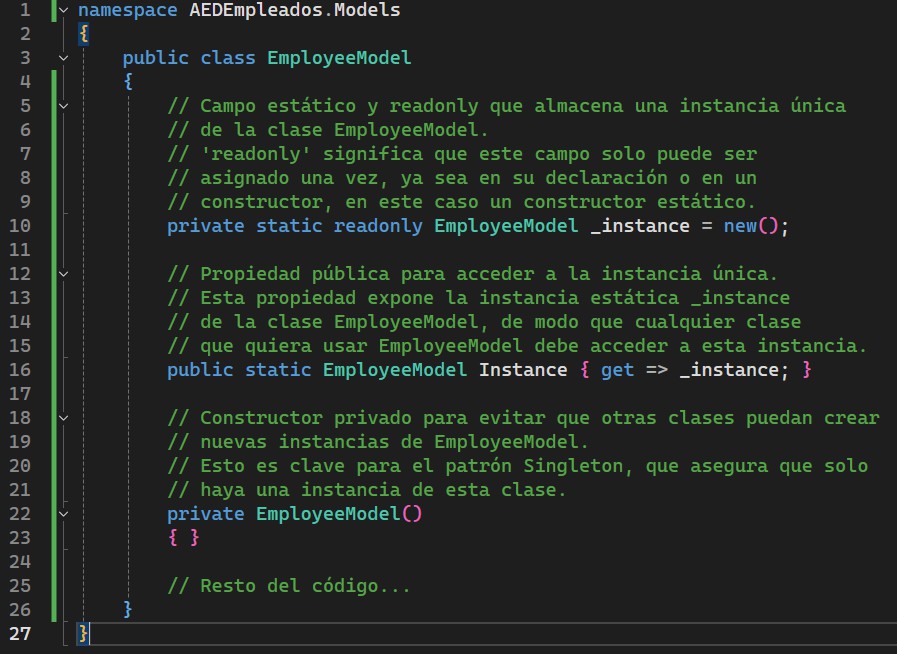
Link: [Static Classes and Static Class Members - C# | Microsoft Learn](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members)

1. **Usar una clase estática separada para almacenar los empleados**

Una última alternativa sería crear una clase aparte que sea estática para que tenga los beneficios explicados en el punto anterior y simplemente desde **EmployeeModel** con una referencia a esta clase manipular el arreglo de **Employee[]**.

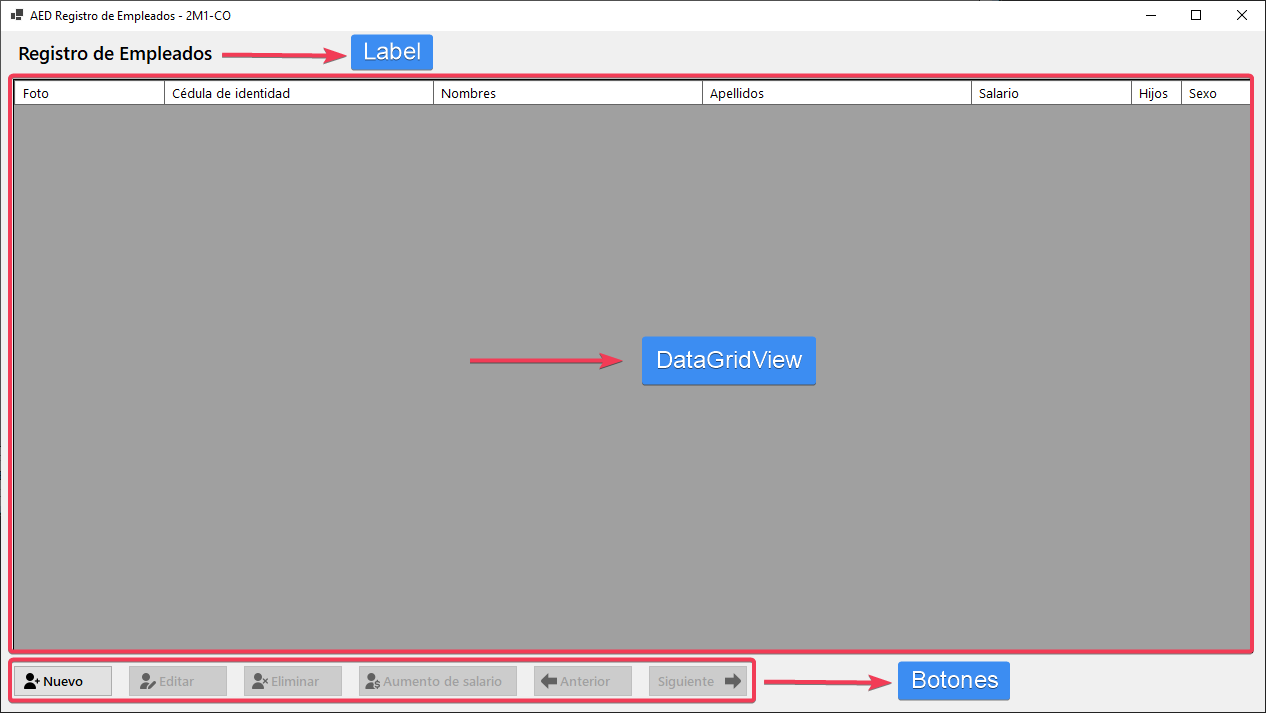
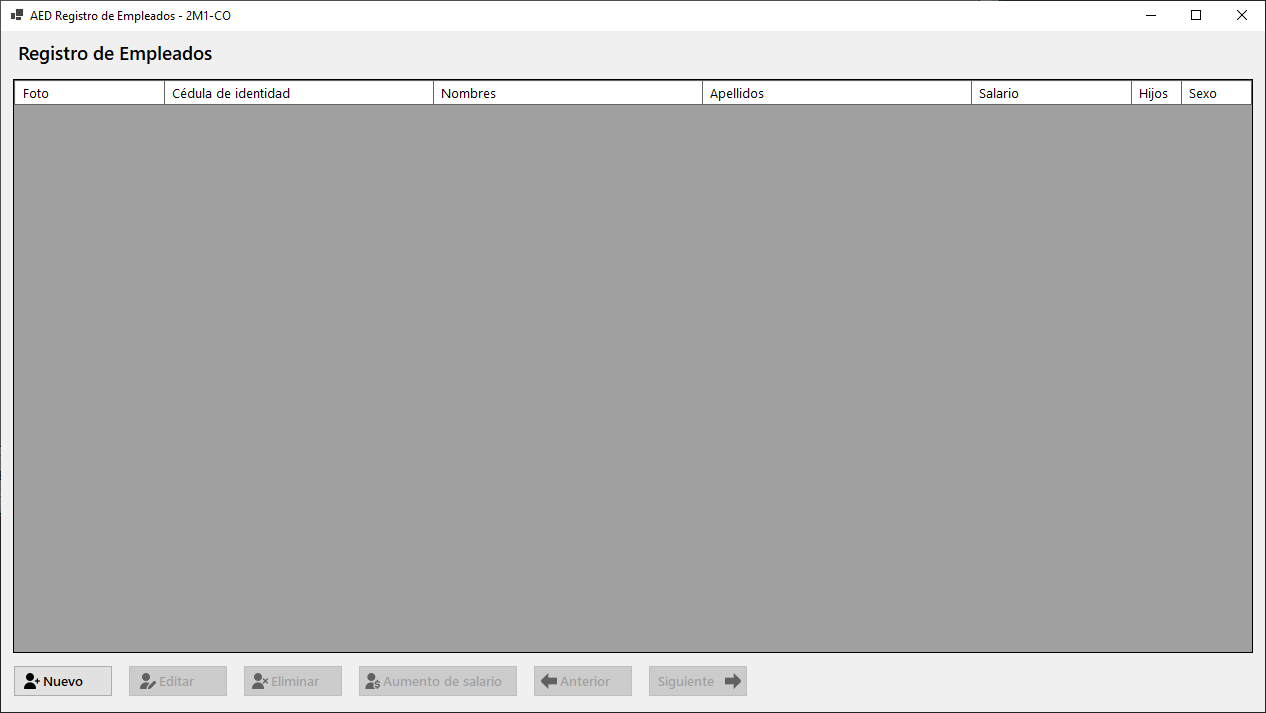
En este caso, implementaremos el patrón de diseño **Singleton** en la clase **EmployeeModel**

para continuar con la guía, para ello hacemos lo siguiente:



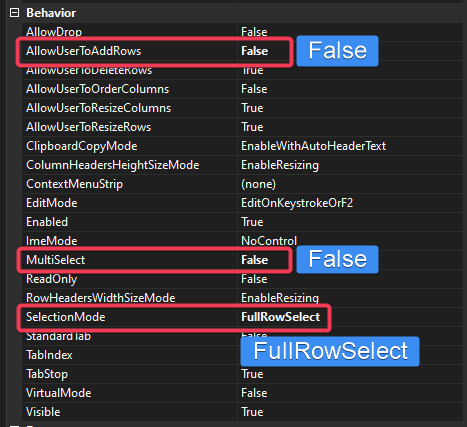
Con esto ya hemos finalizado nuestro modelo de datos.

1. **Diseño del formulario principal o FrmMain**



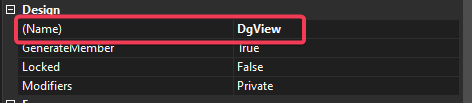
El diseño de cada componente **Label, Buttons** y **DataGridView** se dejan a gusto del estudiante, nos centramos solamente en las propiedades funcionales del **DataGridView** que nos permitirán mostrar datos en ella serían las siguientes:

En el panel de propiedades del **DataGridView**, en la sección **Behavior**

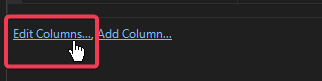


* + **AllowUserToAddRows:** Eliminará la fila vacía que aparece al final de los registros
  + **MultiSelect:** Hará que solamente se pueda seleccionar una fila a la vez
  + **SelectionMode:** Hará que al hacer click sobre cualquier fila de la tabla se seleccione toda la fila y no solo la celda donde se hizo click

El nombre del DataGridView es a gusto, para la guia le llamaremos de la siguiente forma

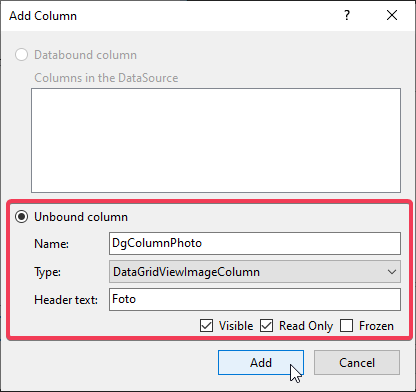


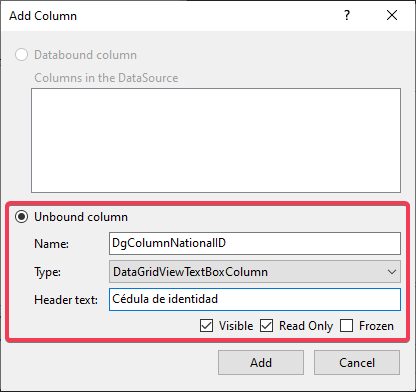
Ahora vamos a definir el número de columnas, el nombre de cada una de ellas, el tipo de dato de cada columna y finalmente la propiedad de nuestra entidad que la utilizara, al final del panel de propiedades tenemos la opción **Edit Columns…**



En la siguiente ventana de diálogo agregaremos cada una de las siguientes columnas:

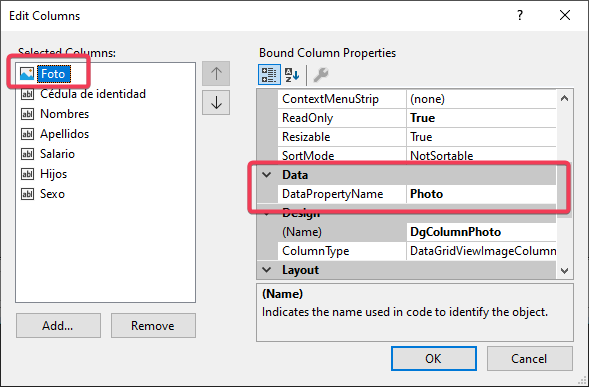
**Columna Foto (Type DataGridViewImageColumn)**

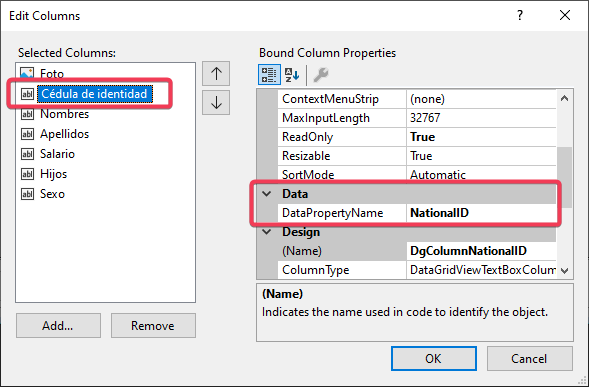


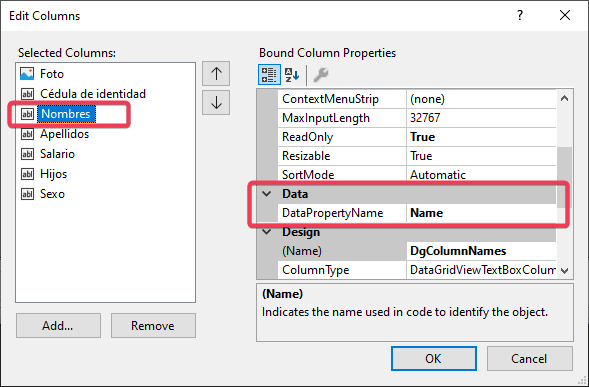
**Columna Cédula de identidad, Nombres, Apellidos, Salario, Hijos, Sexo (Type DataGridViewTextBoxColumn)**

En este caso el **Name** de la columna no es importante para vincularla a nuestra entidad **Employee**, es solo el nombre de la columna en el archivo **.cs** a continuación veremos cómo vincular cada columna con cada propiedad de la entidad desde el diseñador.

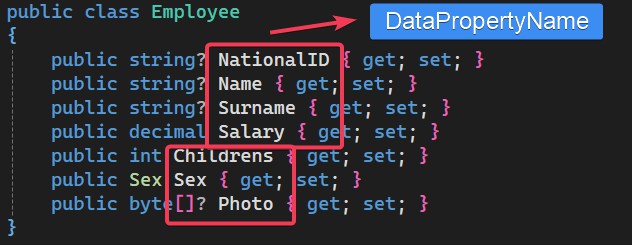
Una vez con las columnas ya agregadas, procedemos a buscar en las propiedades de cada columna la que dice **DataPropertyName** y ahí pondremos el nombre de la propiedad de nuestra entidad que queremos que utilice esa columna (a continuación en la imagen)





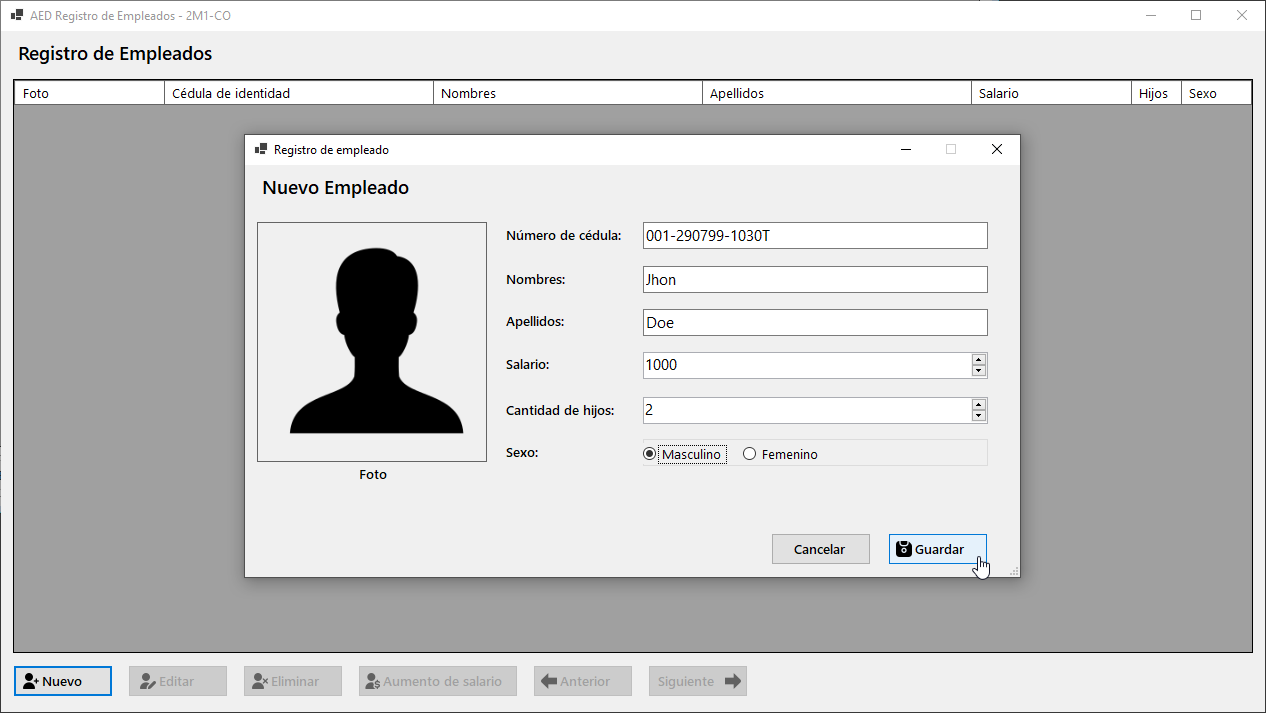


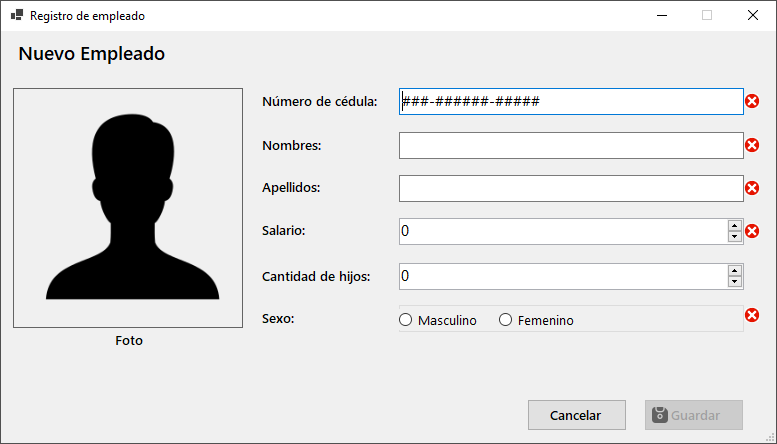
Debemos hacer lo mismo con cada una de las siguientes columnas restantes **Apellidos, Salario, Hijos y Sexo.**

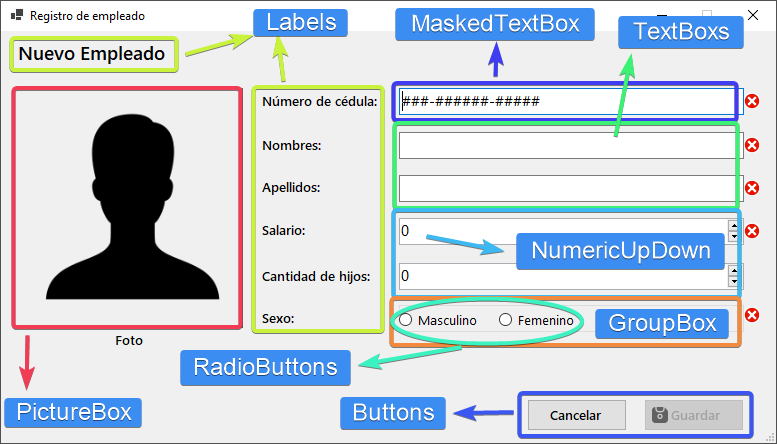


Bien, ahora a los botones les asignaremos un nombre y posteriormente su evento click desde el panel de propiedades.

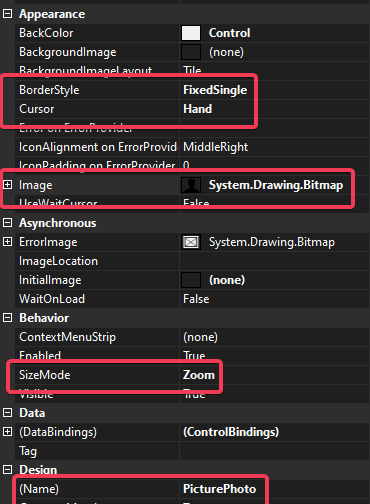
1. **Diseño del formulario FrmEmployee**



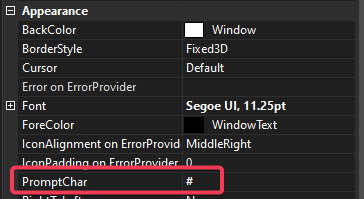


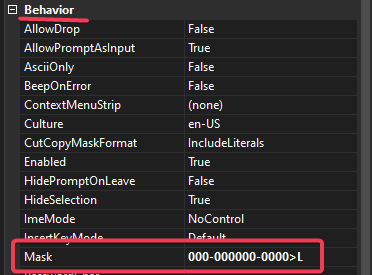


**Propiedades del PictureBox (Foto)**



**Propiedades de MaskedTextBox (Cédula de identidad)**





La máscara se utiliza para restringir y validar el formato de la entrada de datos mientras el usuario escribe. La siguiente máscara se encarga de validar el formato de la cédula

**000-000000-0000>L**

**0: Digito 0-9**

**>: Todo lo que este despues lo pondrá en mayúscula L: Una letra a-z A-Z**

Para ver una lista completa de caracteres disponibles y su significado al crear una máscara: [MaskedTextBox Class (System.Windows.Forms) | Microsoft Learn](https://learn.microsoft.com/en-us/dotnet/api/system.windows.forms.maskedtextbox?view=windowsdesktop-8.0)

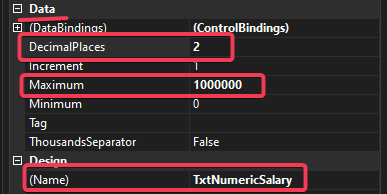
[MaskedTextBox.Mask Property (System.Windows.Forms) | Microsoft Learn](https://learn.microsoft.com/en-us/dotnet/api/system.windows.forms.maskedtextbox.mask?view=windowsdesktop-8.0)

**Propiedades de los TextBox (Nombres y Apellidos)**



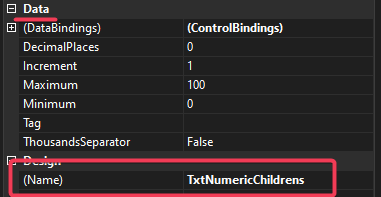
**Propiedades del NumericUpDown (Salario)**

(El máximo y mínimo son valores arbitrarios, en este caso va de 0 a 1,000,000)



**Propiedades del NumericUpDown (Hijos)**

(El máximo y mínimo son valores arbitrarios)



**Propiedades del GroupBox que contiene los RadioButtons**



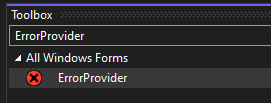
**Propiedades de los RadioButtons (Masculino y Femenino)**

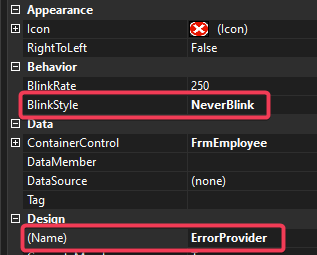


**Propiedades de los Button (Cancelar y Guardar)**



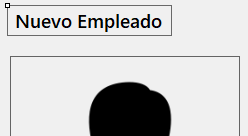
Ahora para poder agregar el indicador de error a la par de cada componente, debemos arrastrar un **ErrorProvider** sobre el formulario en cualquier parte, este no se le da una ubicación desde el diseñador si no desde código programaremos la lógica para que se muestre en cada componente.



**Propiedades del ErrorProvider**

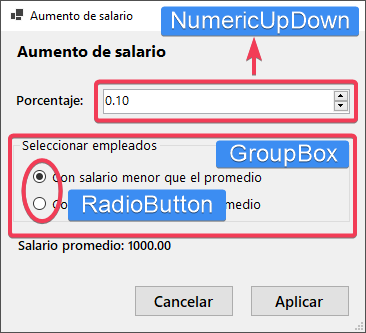
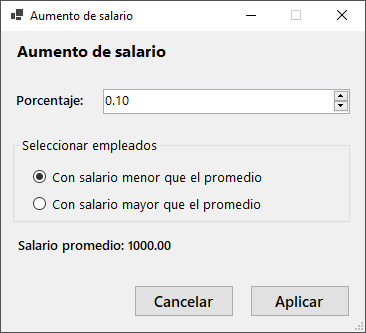
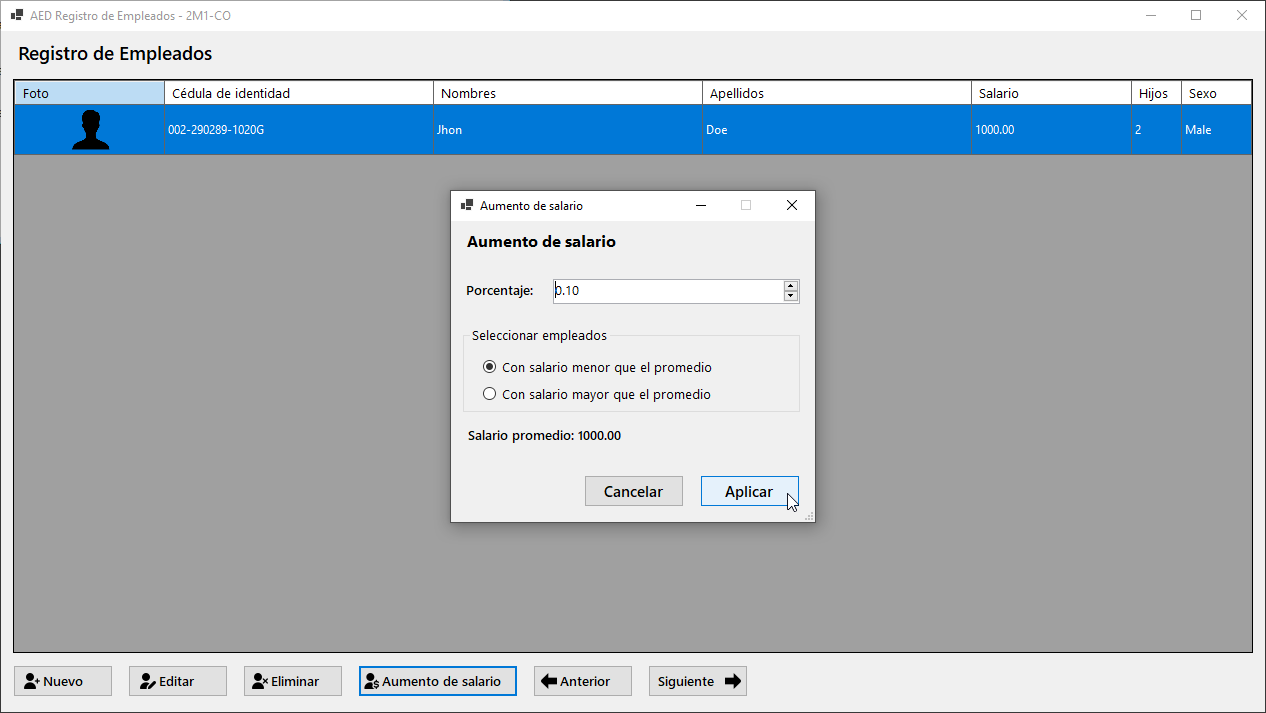
**\*** Este no aparece aún después de modificar sus propiedades la imagen de la derecha es para ilustrar cual es el ErrorProvider y como debe de quedar al final.

**Propiedades del Label principal del formulario**

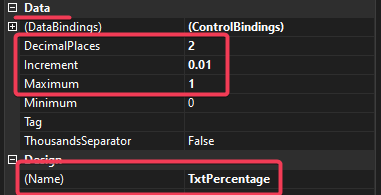


El texto cambiará dependiendo el modo en que se utilice el formulario, ya sea para registrar nuevos empleados o para editar empleados ya existentes.

1. **Diseño del formulario FrmEmployeeSalary**



**Propiedades del NumericUpDown(Porcentaje)**



**Propiedades de los RadioButtons**



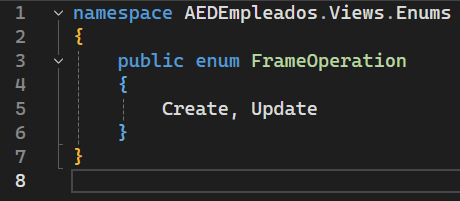
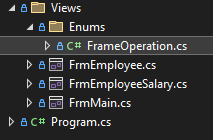
**Propiedades de los Botones (Cancelar y Aplicar)**



**Propiedad del Label (Salario promedio)**

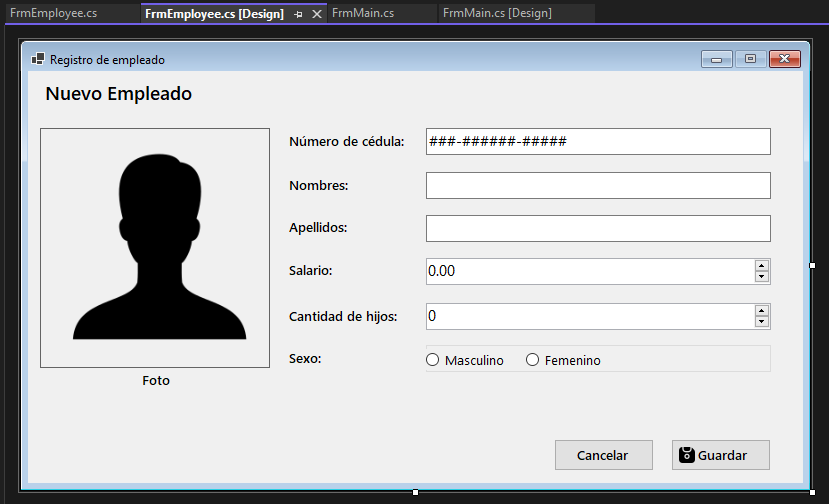


1. **Creación de enums de tipo FrameOperation en carpeta Views > Enums**

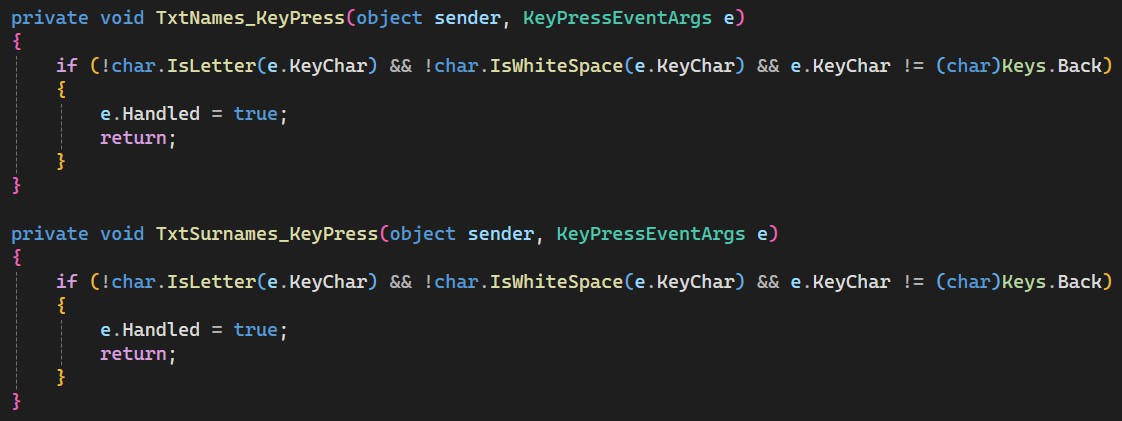


Por medio de este enum podremos indicar al **FrmEmployee** si queremos agregar un nuevo empleado o si queremos editar la información de uno existente cuando lo vayamos a mostrar.

1. **Lógica del formulario FrmEmployee**



**Evento KeyPress de los TextBox Nombres y Apellidos**



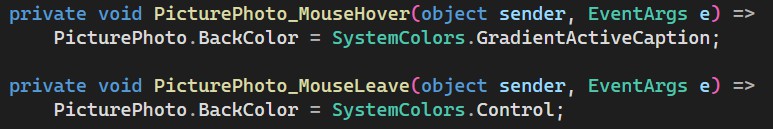
Se valida cada vez que se pulsa una tecla para que sea una letra, espacio en blanco o Backspace (tecla eliminar).

**Evento click del botón Cancel**

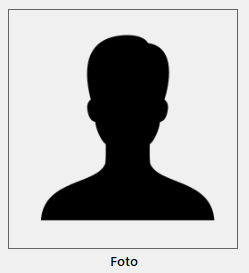
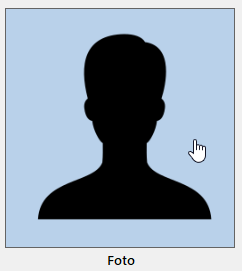


Simplemente cerramos el formulario **FrmEmployee**

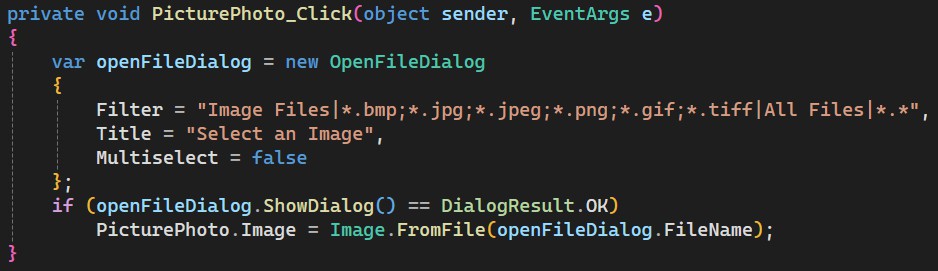
**Evento MouseHover y MouseLeave del PictureBox**



Se cambia color de fondo del picturebox cada vez que se pasa el mouse sobre el componente

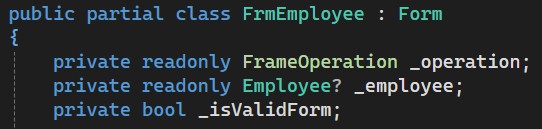


**Evento click del PictureBox**



Abre el explorador de archivo y filtra imágenes por uno de los formatos soportados por el picture box de Windows Forms, si se selecciona una imagen se muestra en el componente.

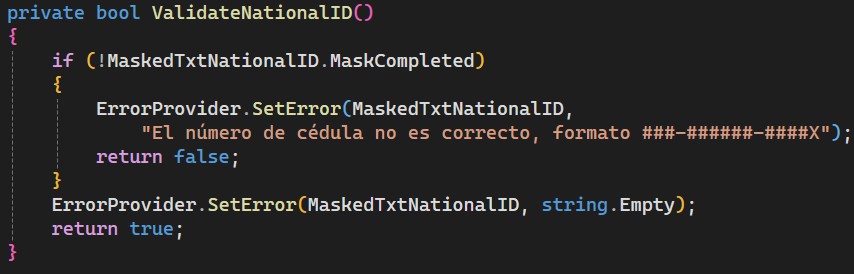
**Variables globales del formulario FrmEmployee**



**Creamos un método para validar la entrada de cada componente apoyandonos del ErrorProvider para dar feedback al usuario**

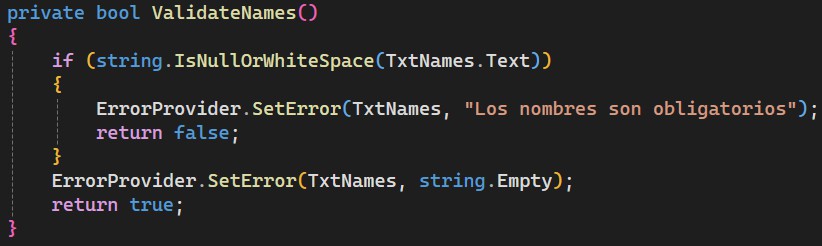
Para utilizar el ErrorProvider sobre un componente, debemos de indicarle cual es el componente y un mensaje de error que acompañe al icono de error

**Validación de cédula de identidad**

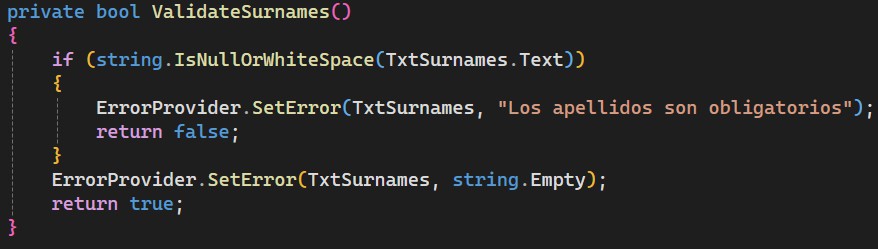


Para validar la cédula de identidad lo hacemos por medio de la propiedad **MaskCompleted** del componente **MaskedTxtNationalID**

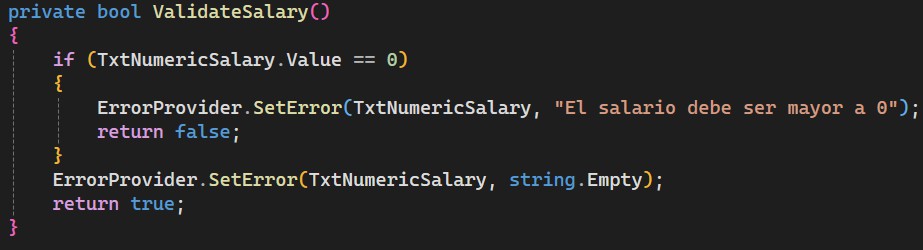
**Validación de los nombres**



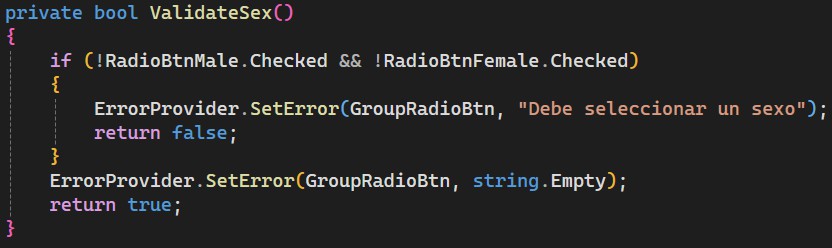
**Validación de apellidos**



**Validación del NumericUpDown salario**



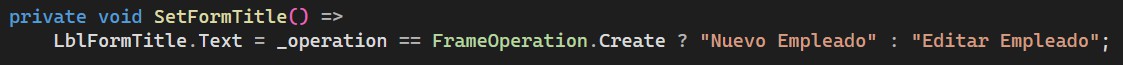
**Validación del GroupBox que contiene a los RadioButtons de Sex**



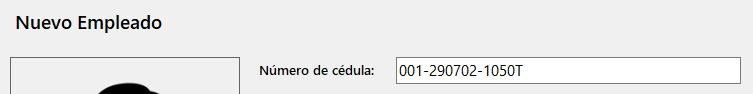
**Validación del NumericUpDown hijos**

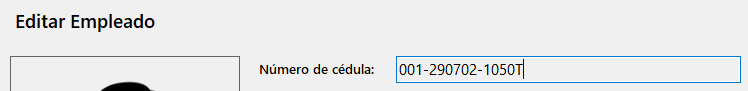
\* Para este campo de texto no se agregó una validación, dejando de forma opcional el agregar una cantidad de hijos, hijos >= 0.

**Método SetFormTitle()**

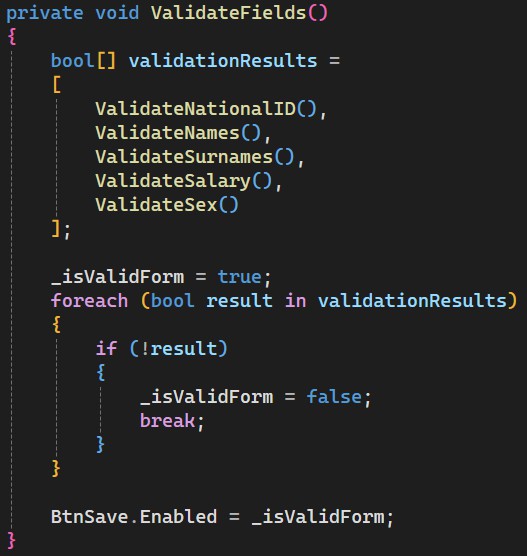


Este método nos ayudará a establecer el nombre del formulario dependiendo el modo en el que se abra, ya sea para registrar un nuevo empleado o para editar uno ya existente.



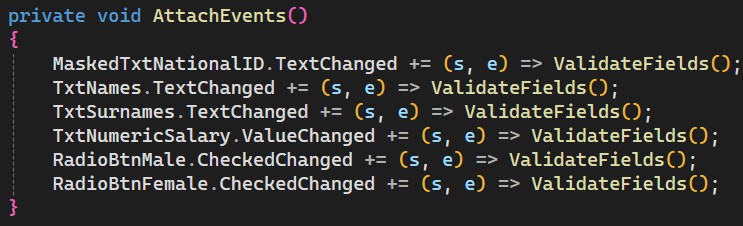


**Método ValidateFields()**



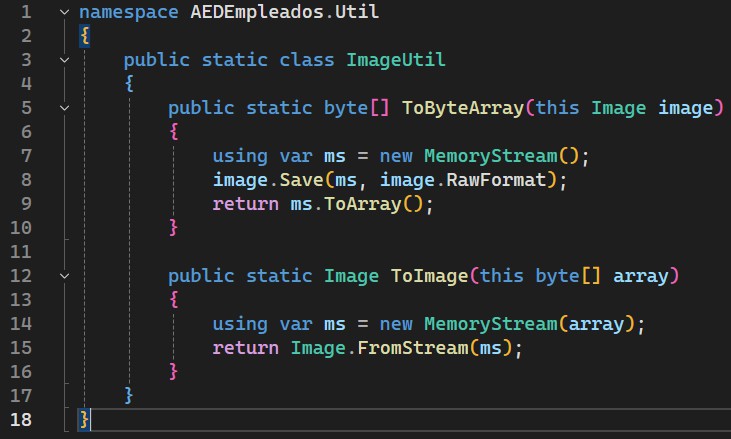
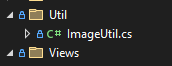
Verificamos si alguna de las validaciones es **false**, si algún componente contiene información incorrecta o está vacío, a parte del **ErrorProvider** que le hará saber al usuario el botón de **Guardar** se deshabilita y se habilitará nuevamente hasta que todos los componentes pasen las validaciones y haya en ellos la información en el formato solicitado.

**Método AttachEvents()**



Desde código agregamos el evento **TextChanged** para todos los campos de texto y el evento **CheckedChanged** de los radio buttons, vamos a asignarles a todos el método **ValidateFields()** que creamos anteriormente para que se ejecute cada vez que cambie el texto de los campos de texto y al momento que se marque o se desmarque un radio button por otro.

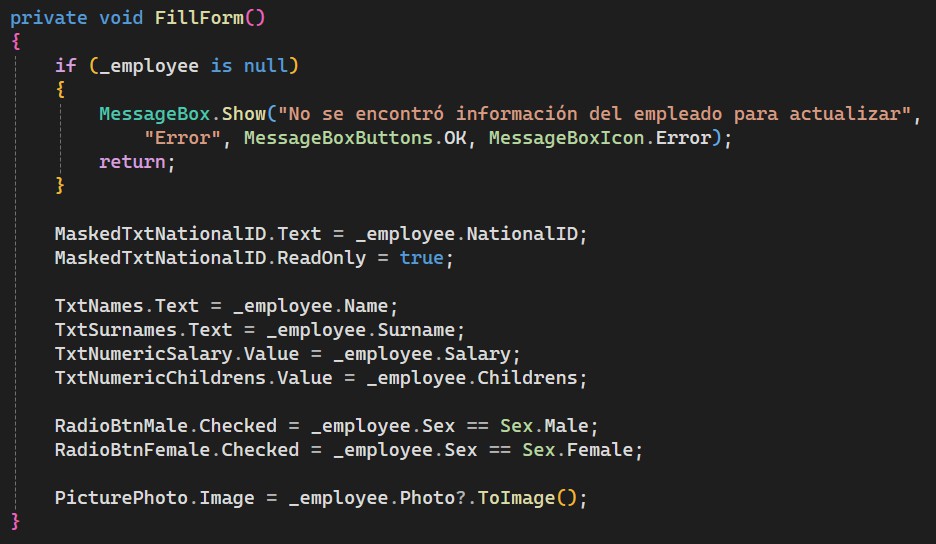
**Crear clase ImageUtil en la carpeta Util**



La clase **ImageUtil** contiene dos **Extension methods** para convertir de **Image** a **byte[]** y viceversa. Los extension methods son métodos que se utilizan para agregar nuevos métodos a tipos existentes sin modificar directamente su código fuente ni crear una clase derivada. Esto es útil cuando se quiere agregar funcionalidad adicional a una clase que no se puede o no se desea modificar.

[Extension Methods - C# | Microsoft Learn](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods)

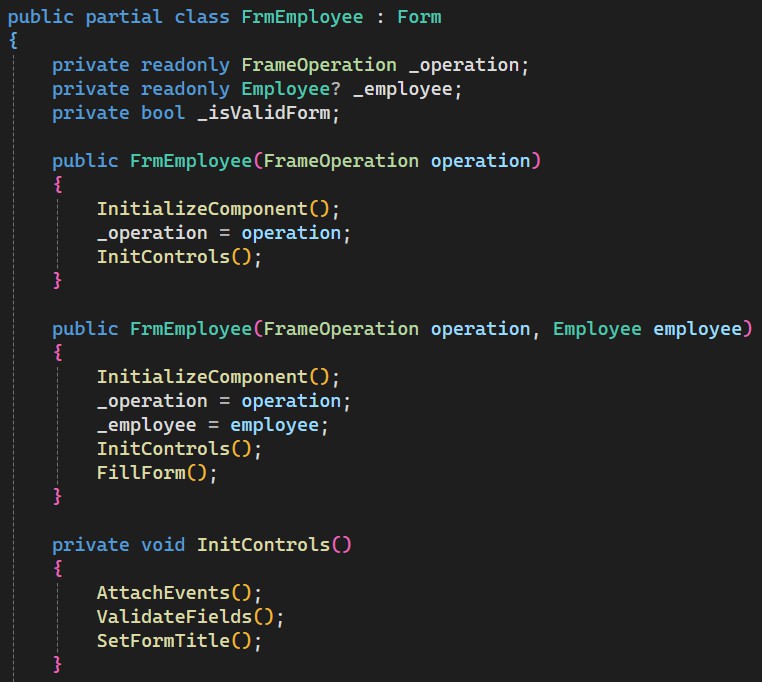
**Método FillForm()**



Este método se encargará de rellenar el formulario apoyándose de la instancia global

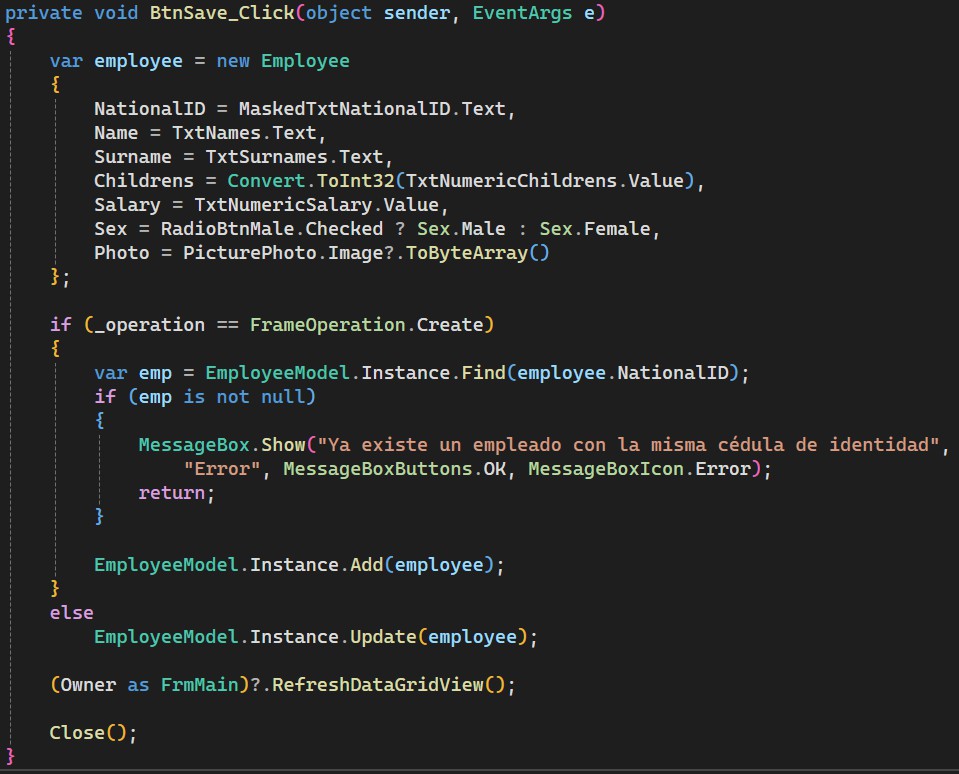
\_employee que es donde se almacenará el empleado que recibiremos desde el formulario principal para editar y a su vez del método **ToImage()** de la clase **ImageUtil** para agregar la imagen al picturebox. También es importante notar que el campo **Cedula de identidad** se establece primero su valor y luego se vuelve solo de lectura, **ReadOnly = true** esto se hace para evitar que se pueda modificar el número de cédula del empleado ya que sería su clave primaria para realizar operaciones sobre el.

**Constructores del formulario FrmEmployee**



Se crean dos constructores, el primero de ellos se utilizara al momento de crear un nuevo registro principalmente y el segundo de ellos se utilizara al momento de editar la información de un empleado existente, también creamos el método **InitControls()** para invocar varios de los métodos que creamos anteriormente una vez se crea el formulario.

**Evento click del botón Guardar**



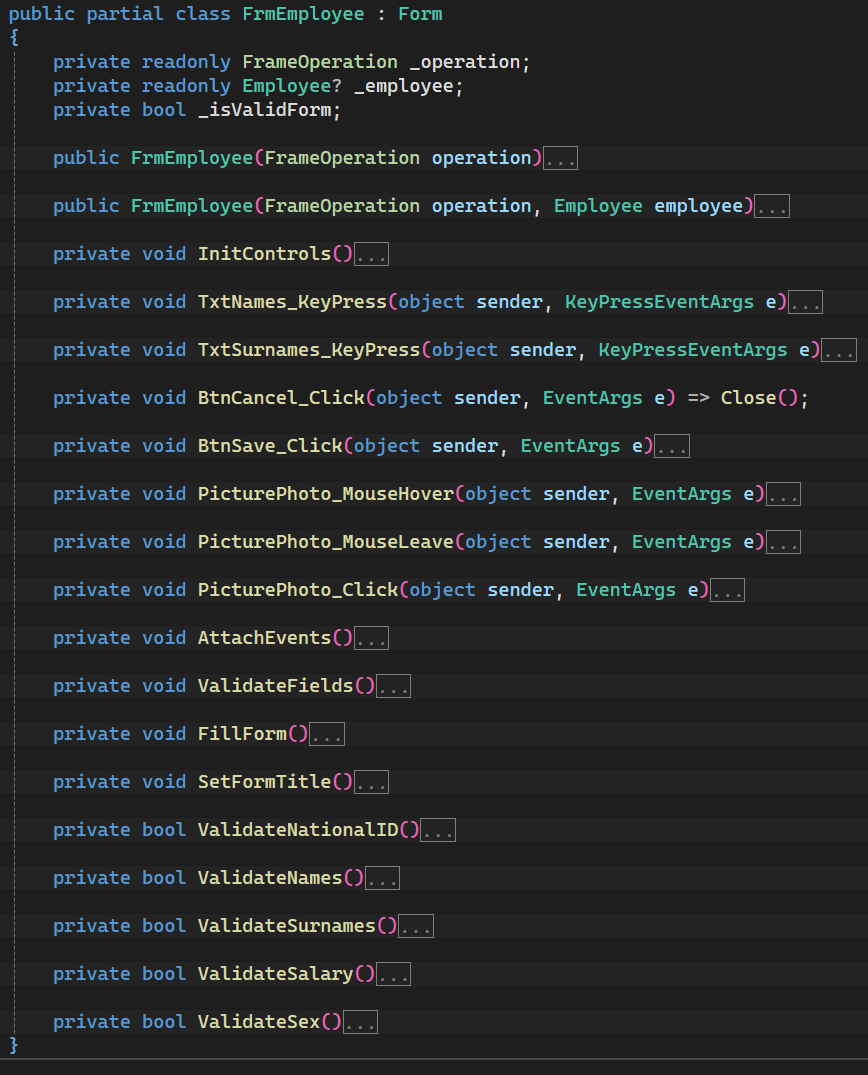
Para guardar o editar un empleado, primero tomamos la información de cada uno de los componentes y no hace falta agregar validaciones en esta parte ya que recordemos que se podrá hacer click solo cuando el formulario contenga los datos en el formato solicitado y ademas el boton ya este habilitado, asi que solo se toman los datos del empleado y se comprueba:

1. **Modo Create:** Se comprueba si ya existe un empleado con el mismo número de cédula, si existe se le indica al usuario, en caso contrario se procede a guardar el empleado haciendo uso de nuestro modelo de datos **EmployeeModel**
2. **Modo Update:** Se actualiza la información del empleado por medio del modelo de datos

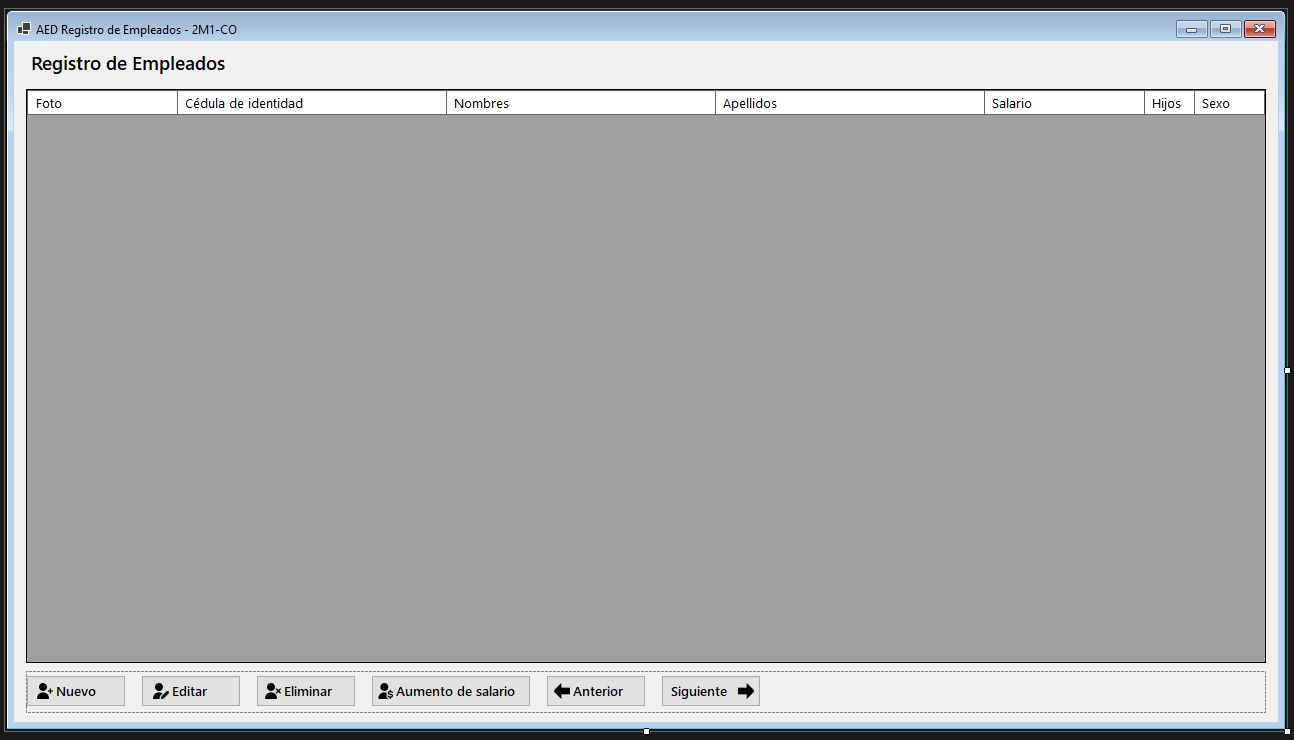
**EmployeeModel**

Finalmente se invoca el método **RefreshDataGridView()** del **FrmMain** que crearemos a continuación para que se actualice el modelo de datos de la tabla con la nueva información, posteriormente se cierra el formulario **FrmEmployee**.

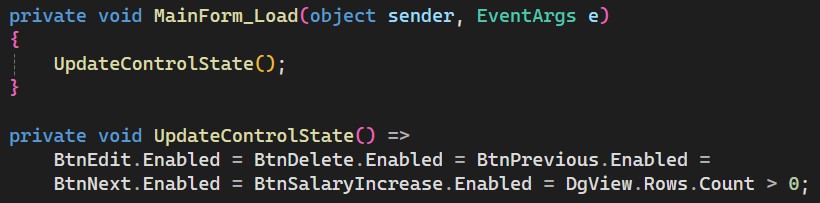
**Resumen de todos los métodos del formulario FrmEmployee**



1. **Lógica del formulario FrmMain**



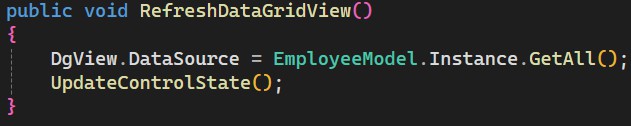
**Agregamos el evento Load() del formulario y el método UpdateControlState()**



El método **UpdateControlState()** se encarga de habilitar o deshabilitar los botones de **Editar, Eliminar, Aumento de salario, Anterior y Siguiente** si no hay filas en la tabla, ya que no tendrían uso alguno mientras no haya registros, una vez se agrega un registro los botones mencionados anteriormente se habilitan.

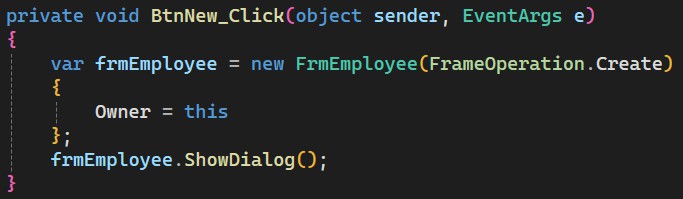
Se invoca por primera vez al cargar el formulario para deshabilitar los botones y que se habiliten una vez se agregue un nuevo registro.

**Método RefreshDataGridView()**



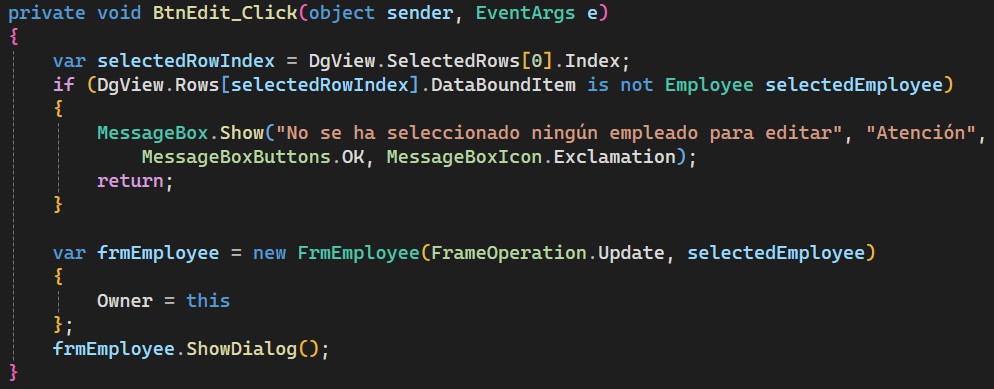
Actualiza el contenido de la tabla trayendo los últimos cambios apoyándose de **EmployeeModel** y finalmente invocando **UpdateControlState()** para ver si tiene que habilitar o deshabilitar los componentes.

**Evento click del botón Nuevo**



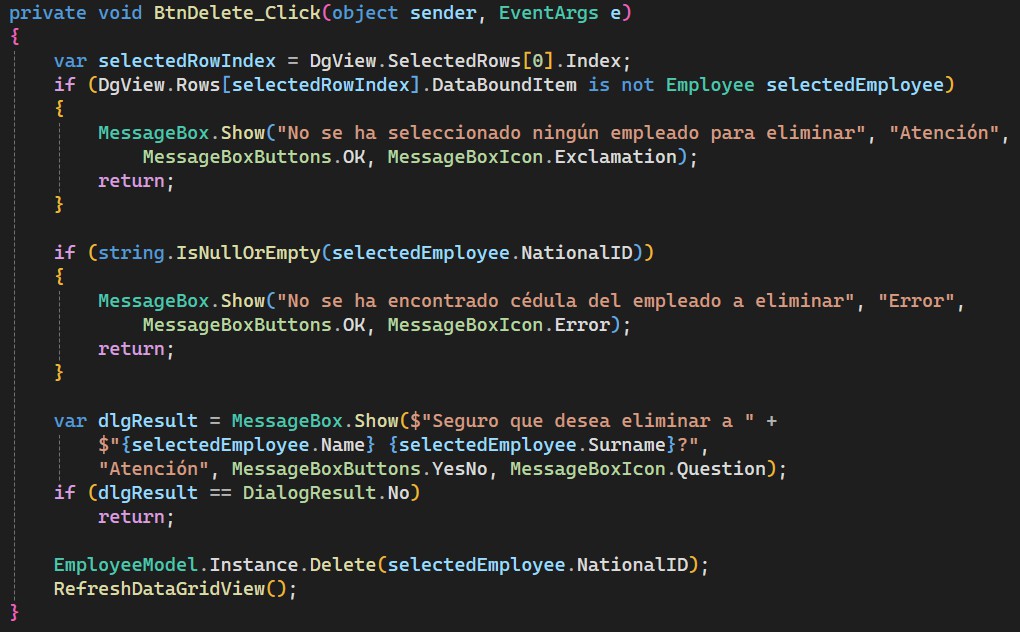
Al pulsar el botón **Nuevo** abrimos el formulario **FrmEmployee** en modo **Create** para agregar un nuevo empleado, también usamos **ShowDialog()** para que la ventana sea modal, es decir que no se pueda hacer click fuera de ella sin antes cerrarla o registrar un nuevo empleado, de esa forma nos aseguramos que no hayan más de un formulario de registro a la vez. De igual forma le indicamos que el **FrmMain** es el **Owner** del formulario **FrmEmployee**, así podremos invocar métodos del formulario **FrmMain** desde **FrmEmployee**.

**Evento click del botón Editar**



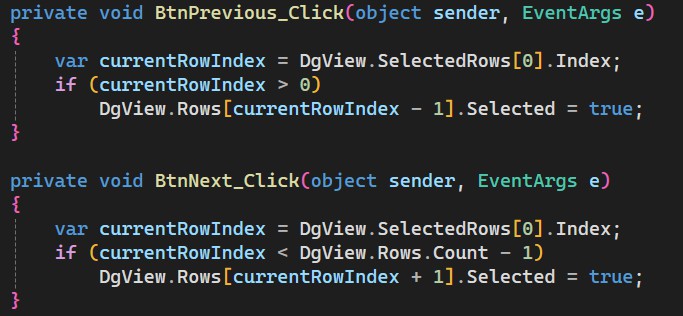
Al pulsar sobre el botón **Editar** se obtiene el empleado de la fila seleccionada y se pasa al formulario **FrmEmployee** en modo **Update**, de igual forma se abre la ventana con **ShowDialog()** para que sea modal.

**Evento click del botón Eliminar**

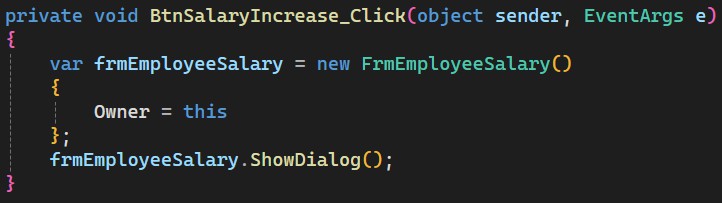


Al pulsar sobre el botón **Eliminar** se obtiene el empleado de la fila seleccionada, preguntamos antes al usuario si está seguro que desea eliminarlo y si la respuesta es distinta de NO procedemos a eliminar el Empleado apoyándonos en **EmployeeModel** y finalmente actualizamos el contenido de la tabla.

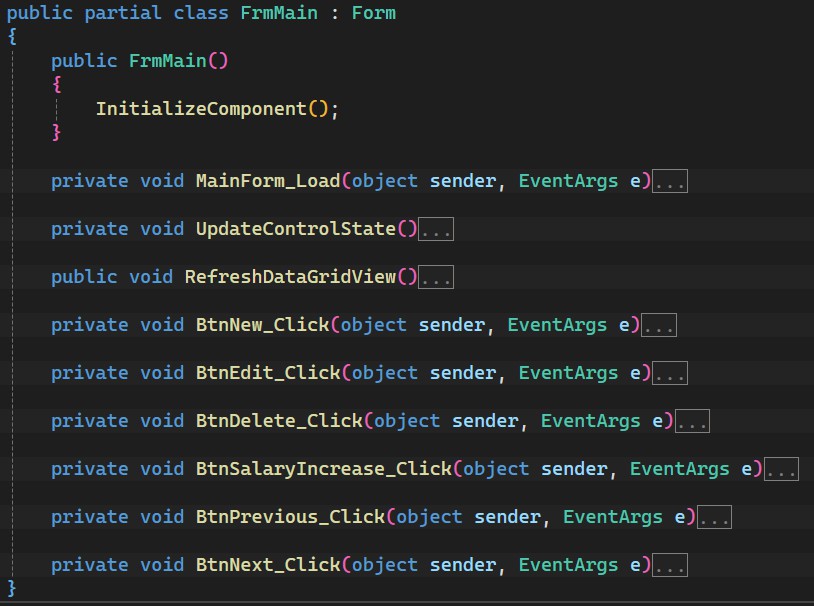
**Evento click del botón Anterior y Siguiente**



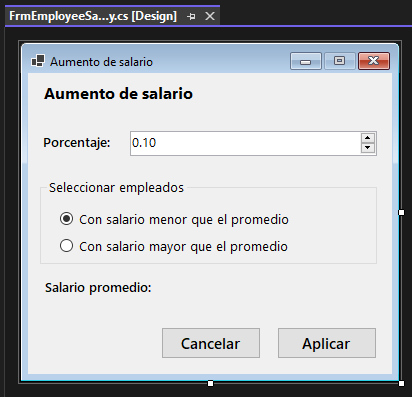
Estos botones se utilizan para navegar por cada fila de la tabla, se toma la fila actual y dependiendo cual se pulsa se aumenta en una posición o se disminuye una posición y luego esa fila se marca como seleccionada.

**Evento click del botón Aumento de salario**

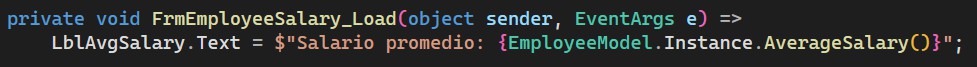
Se abre el formulario **FrmEmployeeSalary()** de forma modal con **ShowDialog()**. **Resumen de todos los métodos del formulario FrmMain**



1. **Lógica del formulario FrmEmployeeSalary**



**Evento Load() del formulario**



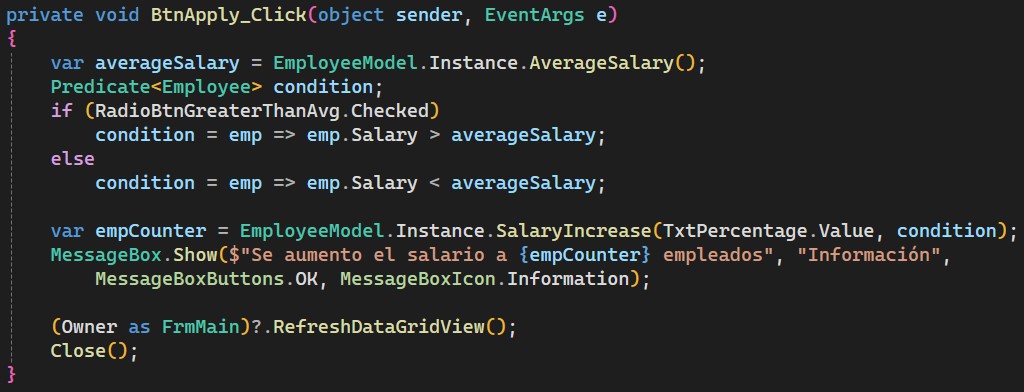
Se obtiene el salario promedio de los empleados apoyándonos en el modelo de datos **EmployeeModel** y el método **AverageSalary()**, asignando el valor al Label encargado de mostrar el salario promedio.

**Evento click del botón Cancelar**

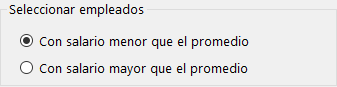


Simplemente cerramos el formulario **FrmEmployeeSalary**

**Evento click del botón Apply**



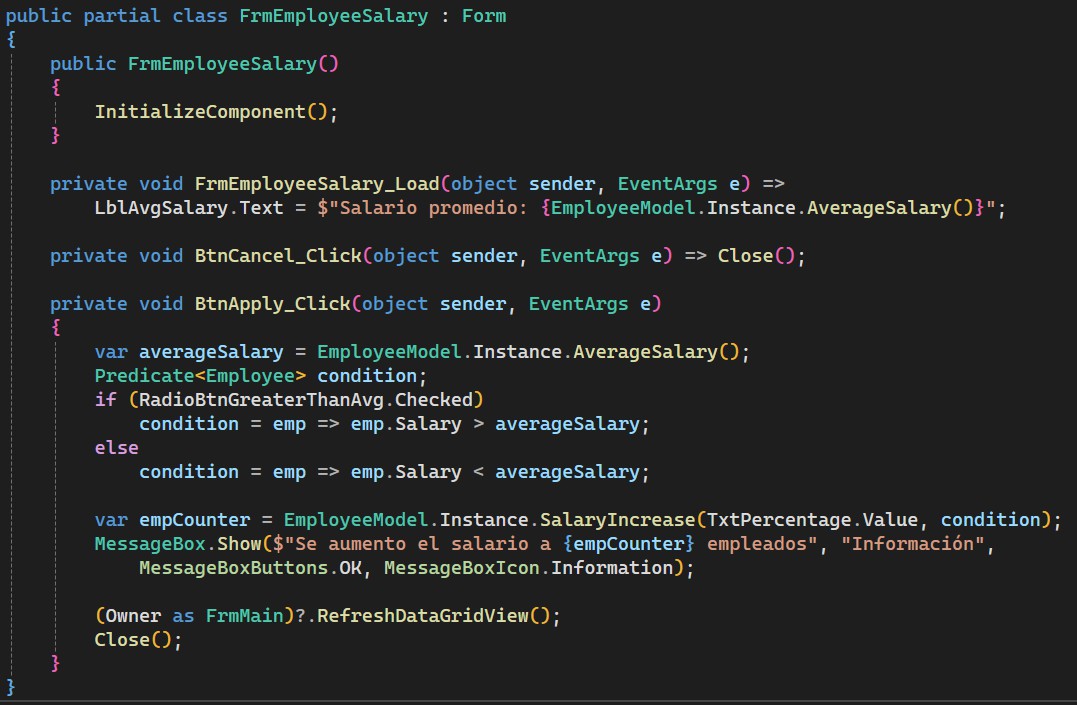
Primeramente se obtiene el salario promedio de todos los empleados en **averageSalary** luego en base al RadioButton que está seleccionado se crea el predicado o condición para la función **SalaryIncrease** de **EmployeeModel**



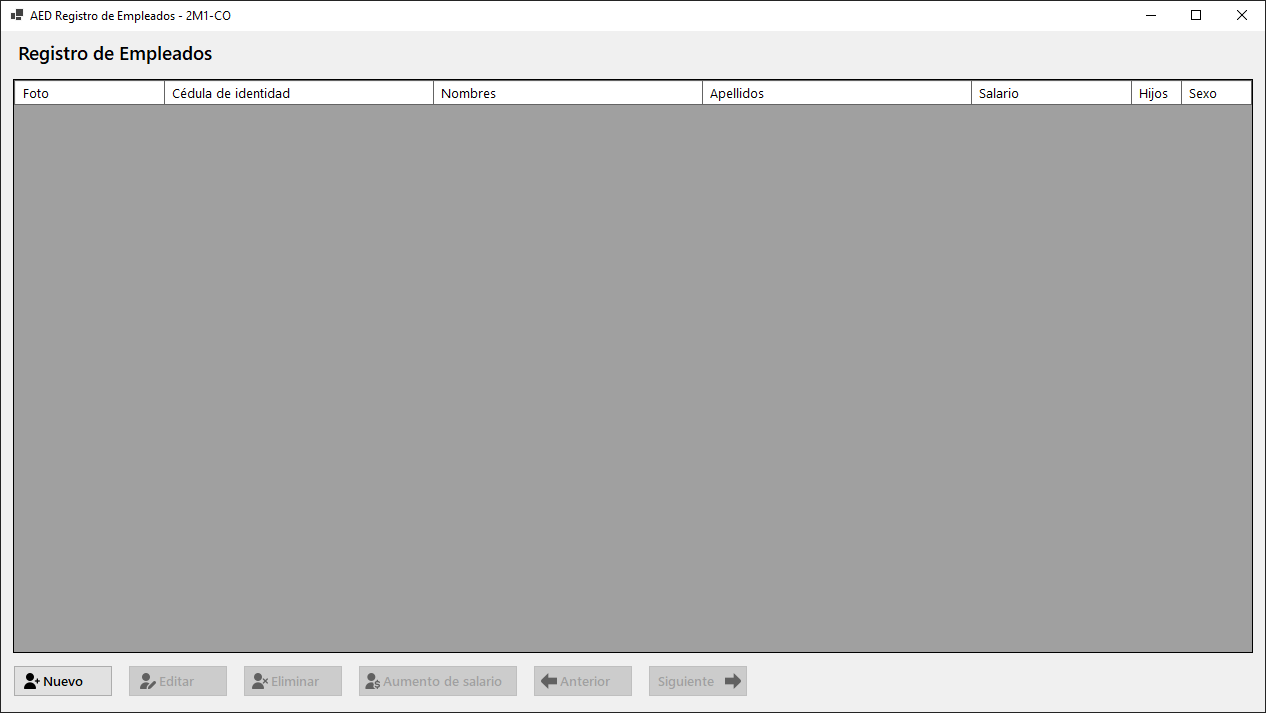
1. El predicado sería Salario del empleado < Salario promedio
2. El predicado sería Salario del empleado > Salario promedio

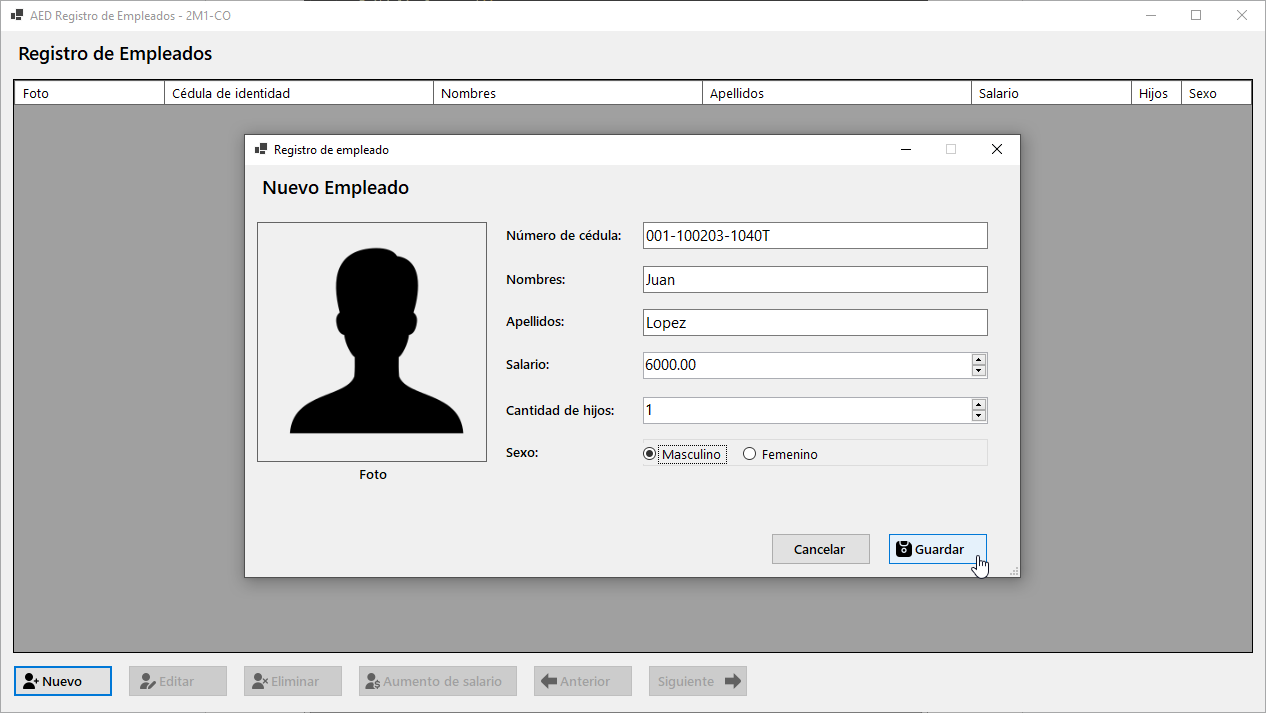
Finalmente se pasa el salario promedio y el predicado a la función **SalaryIncrease** y esta devuelve el número de empleados beneficiado el cual se muestra en un MessageBox, se invoca el método **RefreshDataGridView()** para actualizar el modelo de datos de la tabla y se cierra el formulario de aumento de salario.

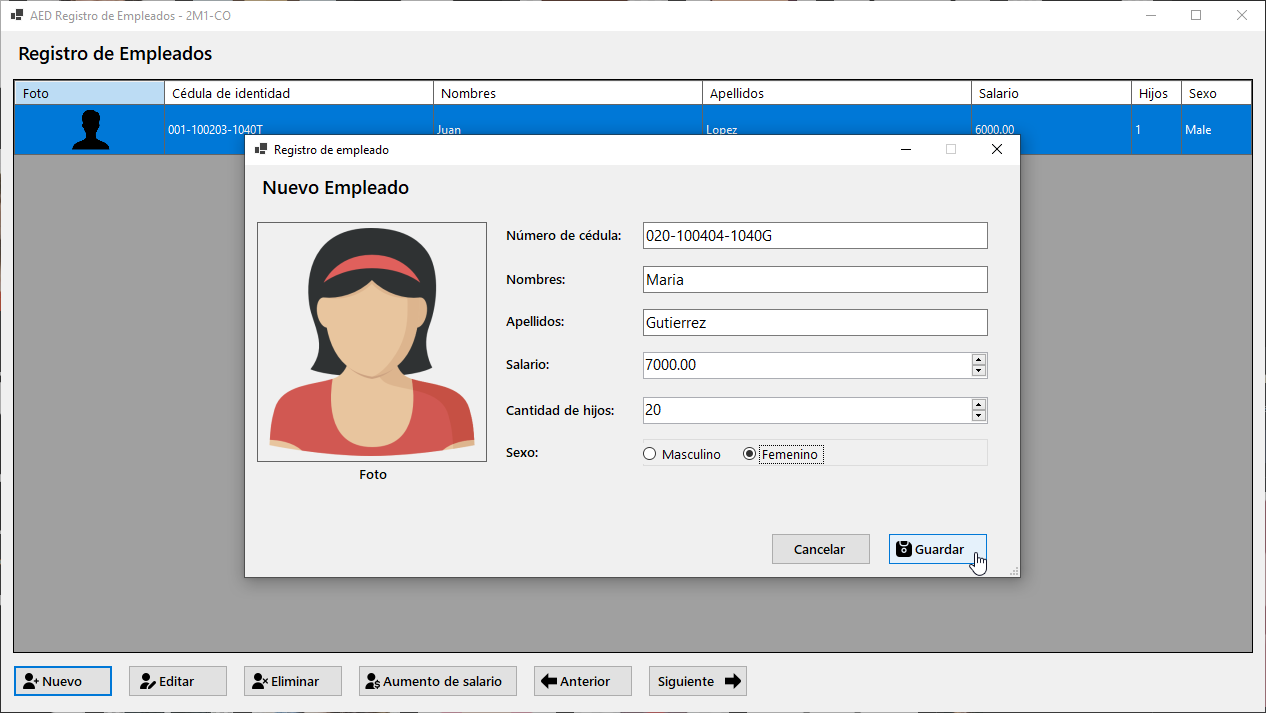
**Resumen de todos los métodos del formulario FrmEmployeeSalary**

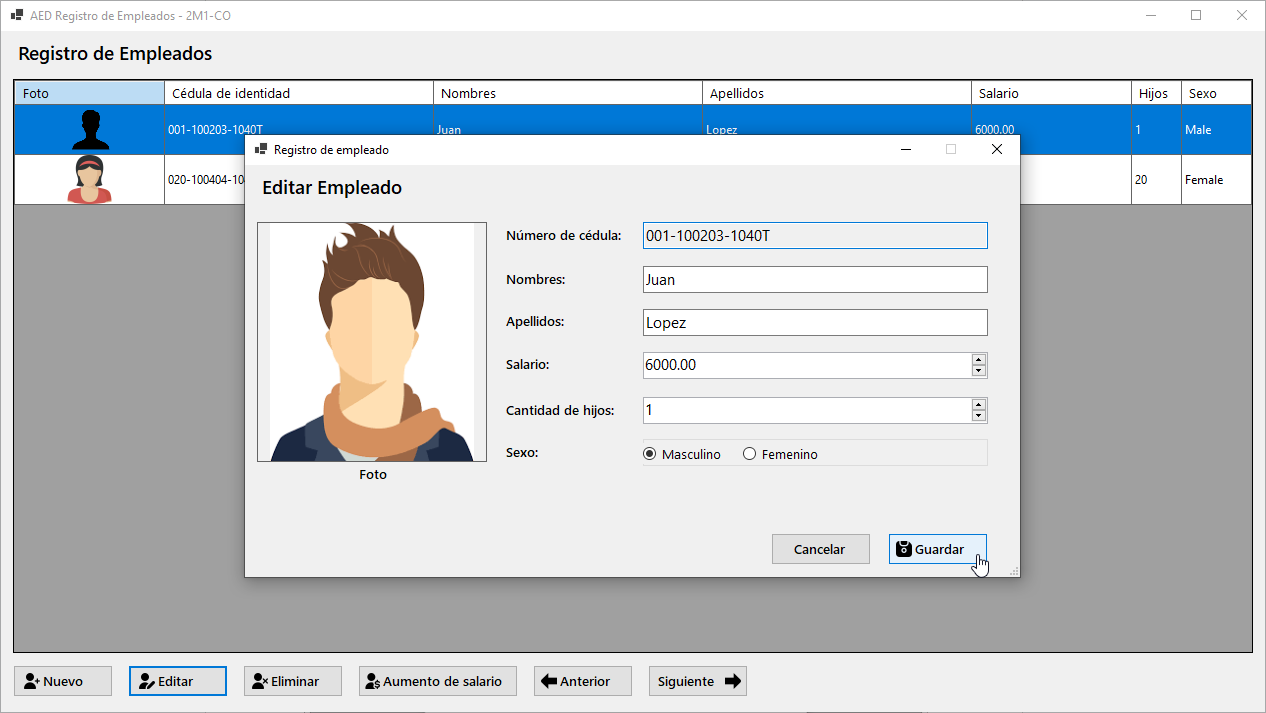


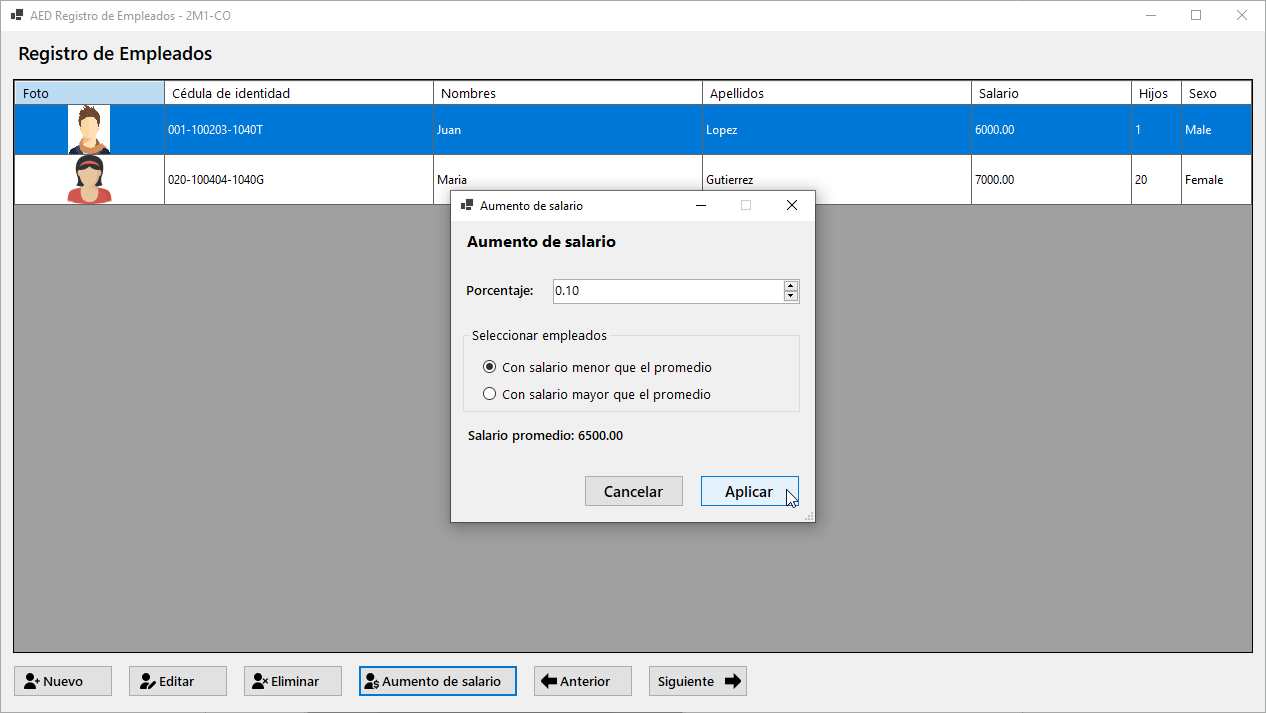
**Ejecución del programa**

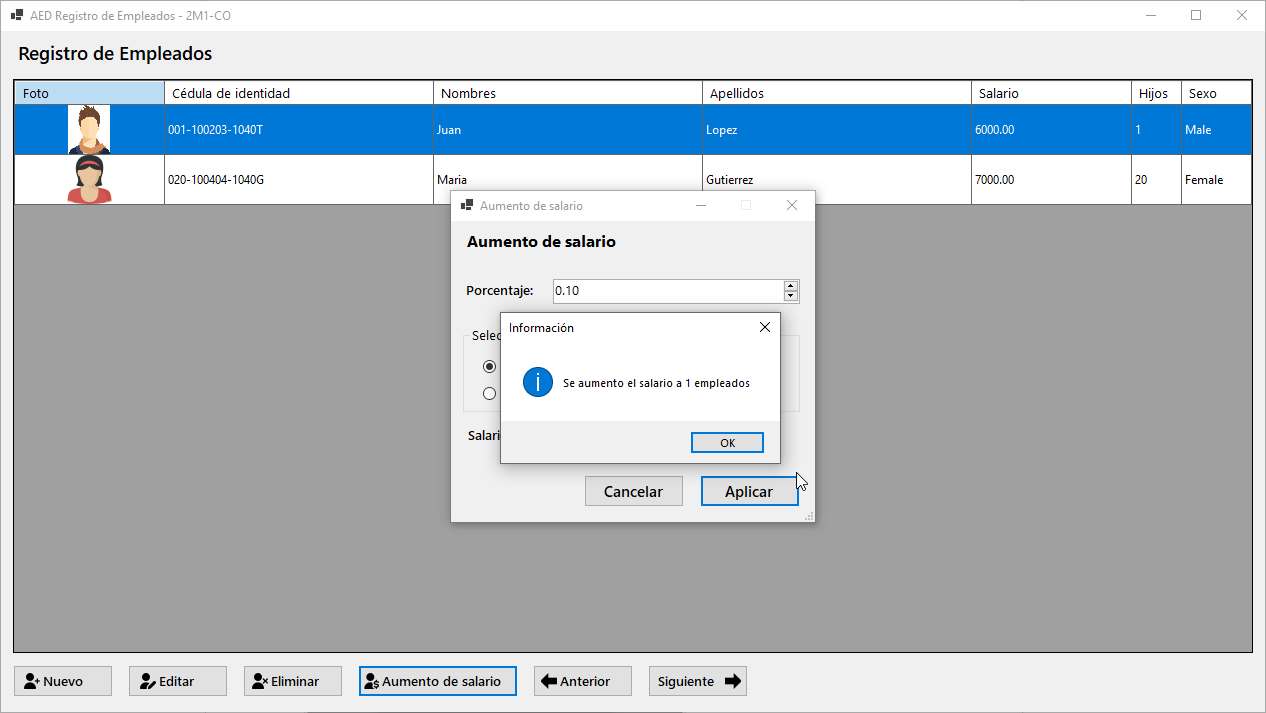


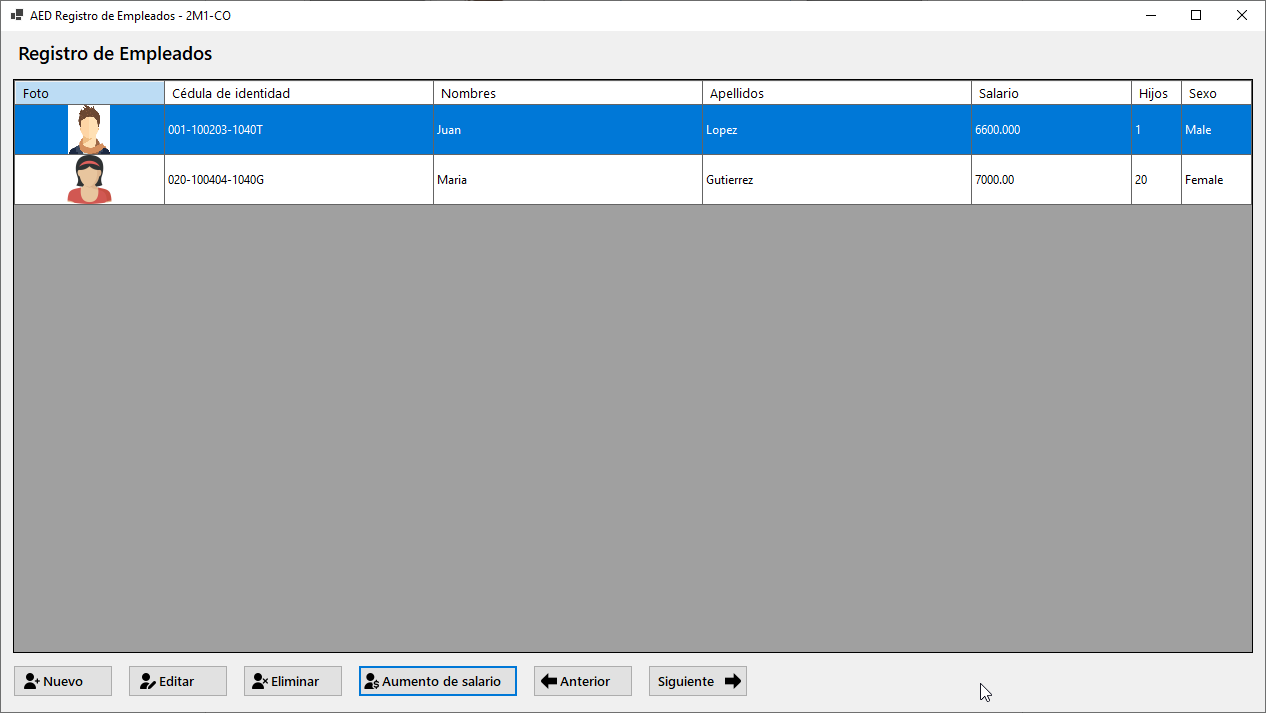












**Resumen de todos los métodos de EmployeeModel**

