ANALISTA PROGRAMADOR

PROGRAMACIÓN II

PRÁCTICA 3: INTERFACES

PROFESOR GONZALO DUARTE

Octubre 2023

Realizado por:
Karen Fernández
Valentina Benítez

Resumen

El proyecto se desarrolla en el lenguaje *C#* a través del *IDE Visual Studio*.
Se solicita realizar para una empresa en crecimiento un repositorio digital de distintos tipos de documentos como facturas, impuestos, recibo de sueldo y remitos, que permita su impresión cuando se necesite. Para abordar este problema se comenzó con la implementación de un diagrama de clases a fin de tener una visión clara y estructurada del mismo. Inicialmente se resuelve mediante un sistema que, a pesar de cumplir su cometido, no resulta escalable y flexible, por lo cual se modifica gradualmente con el fin de aportar estás características y dejar abierta la posibilidad de crecimiento en el futuro acompañando la expansión de la empresa.
En este documento se muestra el proceso de mejora del sistema, que en su fase final implementa los conceptos de herencia, clases abstractas e interfaces.

Contenido

1. Introducción

1.1 Clases y objetos

La clase contiene la definición de las características de un modelo, a las que llamamos atributos o propiedades, junto con la implantación de sus funcionalidades particulares, llamadas métodos. Se puede definir como la abstracción de un objeto. A partir de una clase podemos crear cualquier número de objetos de esa clase determinando así sus características. Podemos pensarlo como si la clase fuera un molde o plantilla y a partir de ese molde creemos distintos ejemplares, aunque no tienen por qué ser todos iguales sino que pueden variar sus atributos. La acción de crear un objeto a partir de una clase se llama instanciar [1].

1.2 Herencia

La herencia permite crear clases que reutilizan, extienden y modifican el comportamiento definido en otra clase. La clase que se hereda se denomina clase base y la clase que hereda se denomina clase derivada. Cuando se define una clase para que derive de otra, la clase derivada obtiene implícitamente lo creado en la clase base, esto quiere decir que reutiliza el código de la clase base sin tener que volver a implementarlo. La clase derivada puede agregar más clases derivadas lo que amplía la funcionalidad de la clase base. [2]

1.3 Interfaces

Las interfaces son un tipo de estructura que define un conjunto de métodos que una clase puede implementar. Se puede decir que las interfaces son un contrato que una clase puede firmar para indicar que utilizará sus funcionalidades. Al implementar una interfaz, una clase debe proporcionar el desarrollo de todos los miembros definidos. Las interfaces se emplean comúnmente para establecer un conjunto de funcionalidades comunes a varias clases, lo que permite una mayor modularidad y reutilización de código. [3]

1.4 Clases abstractas

Las clases abstractas son aquellas que no pueden ser instanciadas, y su finalidad es servir como base para otras clases que sí podrán ser instanciadas. En una clase abstracta alguno de sus métodos está declarado pero no está definido, es decir, se especifica su nombre, parámetros y tipo de devolución pero no incluye código. A este tipo de métodos se les conoce como métodos abstractos. Para que una clase sea considerada abstracta al menos uno de sus métodos tiene que ser abstracto. [4]

1.5 Diagrama de clase

Un diagrama de clases es una estructura estática que se utiliza en la ingeniería de software. Muestra las clases, los atributos, los métodos y la asociación entre ellas, lo que ayuda a desarrollar el código de una aplicación. También se utiliza para describir, visualizar y documentar las diferentes características de un sistema.

Los diagramas de clase son los únicos diagramas UML que pueden asociarse directamente con los lenguajes orientados a objetos y son uno de los diagramas más importantes en la programación. Además, se utilizan para el análisis y el diseño de una aplicación.

Asociaciones

Las asociaciones se utilizan para representar los vínculos que significan la relación estática entre clases. Se dividen en cuatro tipos: asociación unidireccional, bidireccional, agregación y composición.

- Asociación unidireccional: este tipo de asociación se refiere a cuando un objeto contiene otro objeto en su campo. Esta relación significa el flujo de información, la flecha apunta hacia la clase contenida (Figura 1.1).

Figura 1.1: Representación de la asociación unidireccional.

- Asociación bidireccional: se utiliza cuando dos objetos están estrechamente vinculados y pueden almacenarse entre sí en sus campos (Figura 1.2).

Figura 1.2: Representación de la asociación bidireccional.

- Agregación: es un tipo de asociación más específico y muestra la relación "es parte de" entre clases, el diamante hueco se ubica hacia la clase contenedora (Figura 1.3).

Figura 1.3: Representación de la agregación.

- Composición: este tipo de relación se utiliza para representar la dependencia de los objetos de la clase focal. La clase focal contiene objetos, pero los objetos contenidos también se eliminan si se elimina la clase focal, el diamante negro se ubica hacia la clase contenedora (Figura 1.4). [5, 6]

Figura 1.4: Representación de la composición.

2. Desarrollo

Se deja el link en donde se encuentra el código:
https://github.com/KarenFernandez13/ProgramacionII-Practica2-Interfaces/tree/main

Para comenzar con la descripción del proyecto se explicará el siguiente diagrama de clases el cual compone los elementos en los que se basa el trabajo:

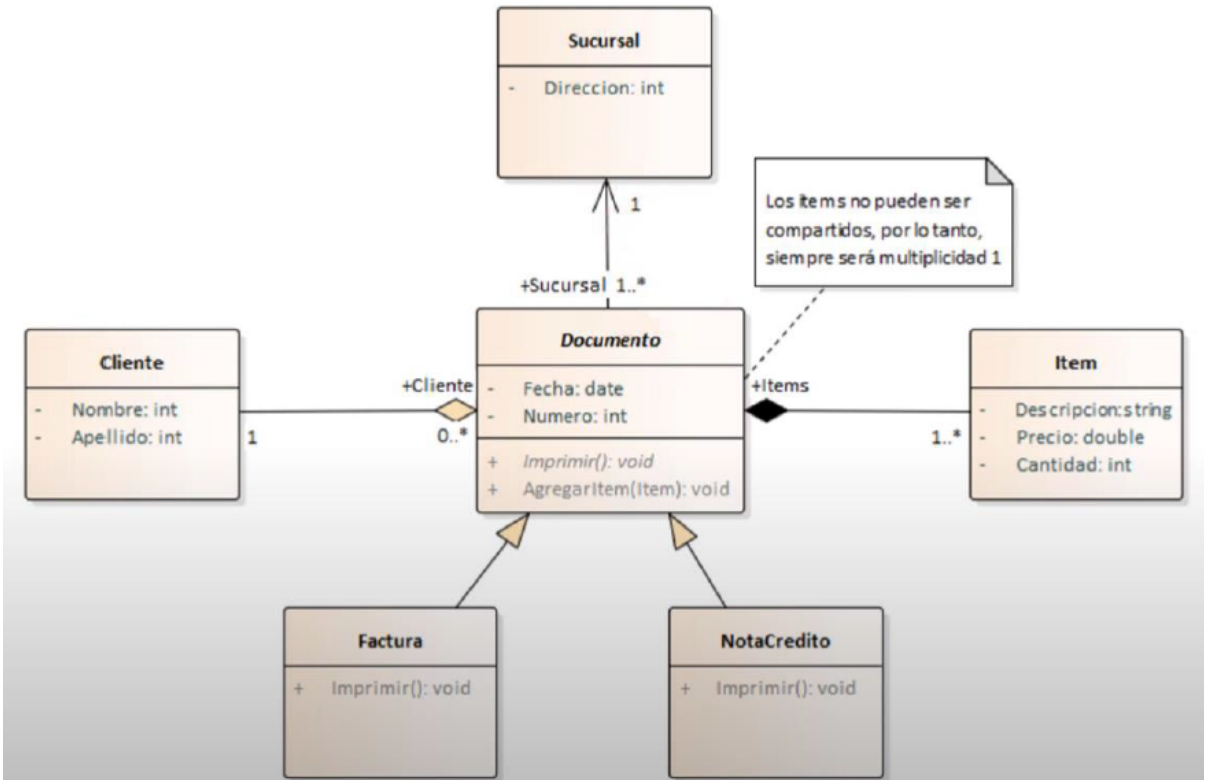

Figura 2.1: Diagrama de clases.

En primer lugar se puede observar en la parte central del diagrama la clase abstracta Documento con sus atributos y métodos, funciona como elemento principal del problema. Luego se encuentran las clases Factura y NotaCredito que cumplen la función de clases derivadas de Documento, esto se representa con flechas en dirección a la clase base. En el sector izquierdo se localiza Cliente con sus atributos, que opera a través de un rombo amarillo con el elemento central, se referencia a una agregación que describe que una clase es parte de otra pero su existencia es independiente, por lo que si una se elimina no requiere consecuencia hacia la otra, se muestra una relación de 1 a 0…*, quiere decir que un cliente puede tener de cero a muchos documentos sin embargo, un documento le pertenece a un cliente. En contraposición se encuentra la clase Ítem con sus atributos y que

se relaciona con Documento a través de un rombo negro, esto representa una composición en donde la vida de Item (componente) depende de la vida de Documento (compuesto), consecuentemente los componentes no pueden ser compartidos con otros compuestos y si se elimina el compuesto también los componentes, se observa una multiplicidad en la que en un documento se pueden encontrar de uno a varios ítems pero cada ítem pertenece a un documento. Por último en la parte superior se ubica la clase Sucursal la cual lleva una relación de 1 a 1…* con Documento, es decir un documento pertenece a una sucursal pero en una sucursal se puede obtener de uno a varios documentos. En este punto se tienen en cuenta solamente los documentos de tipo Factura y Nota de crédito con la función Imprimir incluida en cada uno.

Por consecuencia del crecimiento de la empresa, cada vez se necesitan más tipos de documentos y a su vez se acumulan gran cantidad de estos ya emitidos, por lo cual se decide crear un repositorio para guardarlos digitalmente. Se debe modificar el sistema teniendo en cuenta lo anterior y la posibilidad de imprimir dichos documentos en cualquier momento. Los tipos de documento son: Facturas, Notas de Crédito, Remitos, Recibos de Sueldo, Impuestos Municipales y Factura de Luz (Figura 2.2).

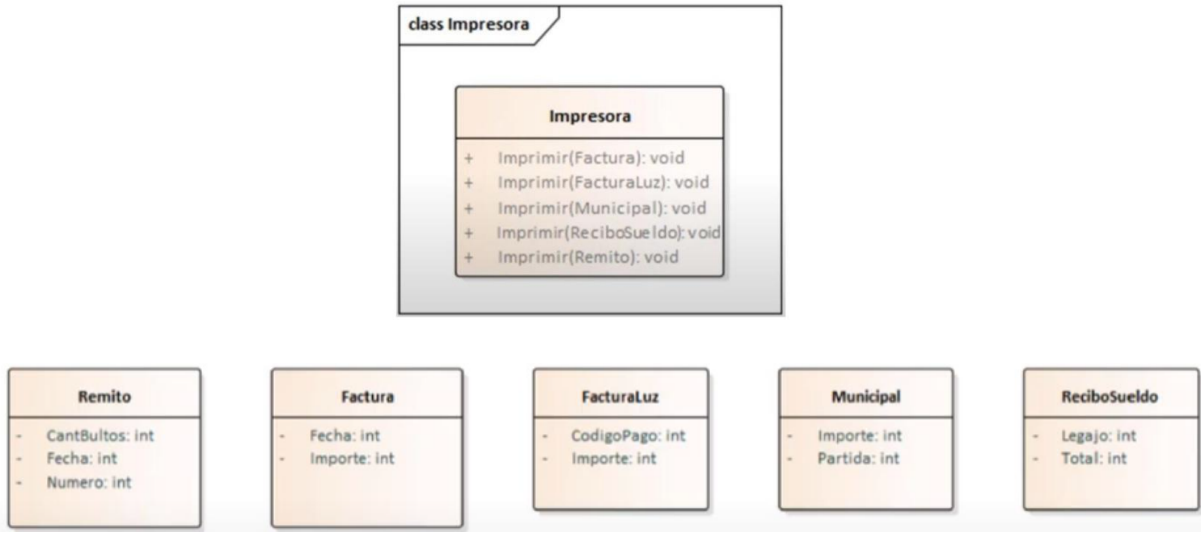

Figura 2.2: Clases con los tipos de documentos que produce la empresa.

Para lograr el desarrollo del código, se piensa en crear las clases correspondientes a cada tipo de documento, con sus atributos, por separado se incluye la clase Impresora, que se compone de varios métodos, cada uno permite la impresión de un tipo de documento, pasándolo como parámetro a dicho método. Posteriormente se notó la posibilidad de mejorar el sistema a través de una mayor abstracción, lo que permitirá el desarrollo de un código más flexible y también más escalable, es decir que deja abierta la opción de agregar

nuevos tipos de documentos en caso de ser necesarios en el futuro. Con este fín, se agregan las clases Documento Contable e Impuestos, ambas abstractas.

Documento Contable cuenta con los atributos Fecha, Importe, Número y Sigla, método Imprimir y además el método abstracto “Total”, que se desarrollará en las clases hijas Factura y Nota de Crédito. Este método es llamado dentro de su clase de orígen, la clase padre, específicamente dentro del método Imprimir, pero viaja a la clase hija, cumple las indicaciones que se le desarrollan allí y devuelve un resultado que forma parte de la construcción del método Imprimir.

La clase Impuesto contiene un solo atributo: Importe. Sus clases hijas, Factura de Luz y Municipal definen un atributo propio para detallar: Código de Pago y Partida respectivamente. Recibo de Sueldo y Remito funcionan como clases independientes con atributos propios (Figura 2.3).

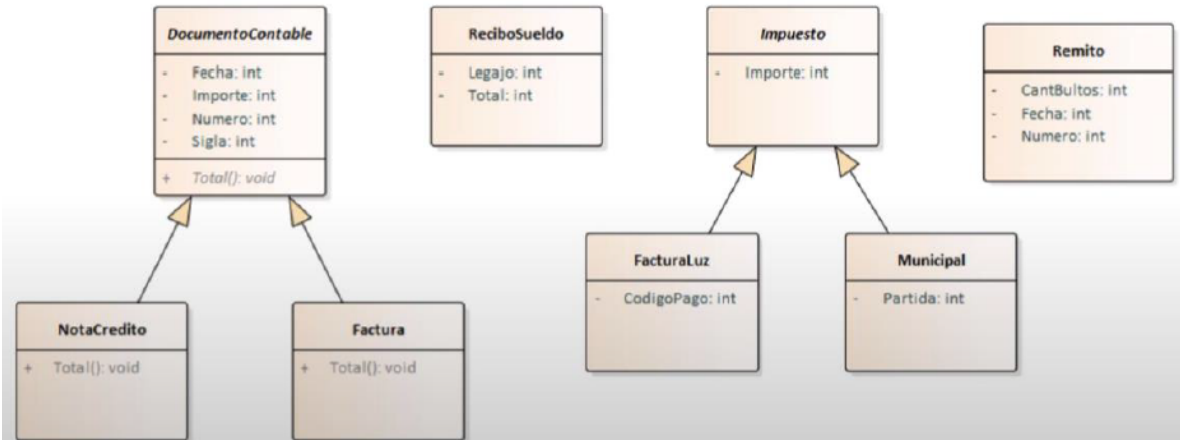

Figura 2.3: Abstracción de tipos documentos.

De esta manera se pueden mejorar los métodos definidos en la clase Impresora, resultando en cinco métodos diferentes según qué documento se quiera imprimir, Documento Contable (permite Factura y Nota de crédito), Factura de Luz, Municipal, Recibo de sueldo y Remito (Figura 2.4).

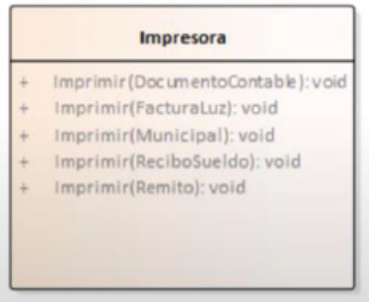

Figura 2.4: Clase que imprime los diferentes tipos de documentos.

Aún así, cada vez que se desee agregar nuevos tipos de documentos, se deberá modificar la clase Impresora para agregar el método que permite imprimirlo, por lo tanto, se modifican las clases nuevamente y se crea un método abstracto Impresora en las clases padres, el cual tendrá que ser obligatoriamente desarrollado por sus clases hijas y, a su vez, en las clases independientes se desarrolla directamente el método Imprimir, ya que éstas no reciben ninguna herencia (Figura 2.5). Ahora la clase Impresora, tendrá 4 métodos Imprimir: Documento contable, Impuestos, Remito y Recibo de sueldo (Figura 2.6).

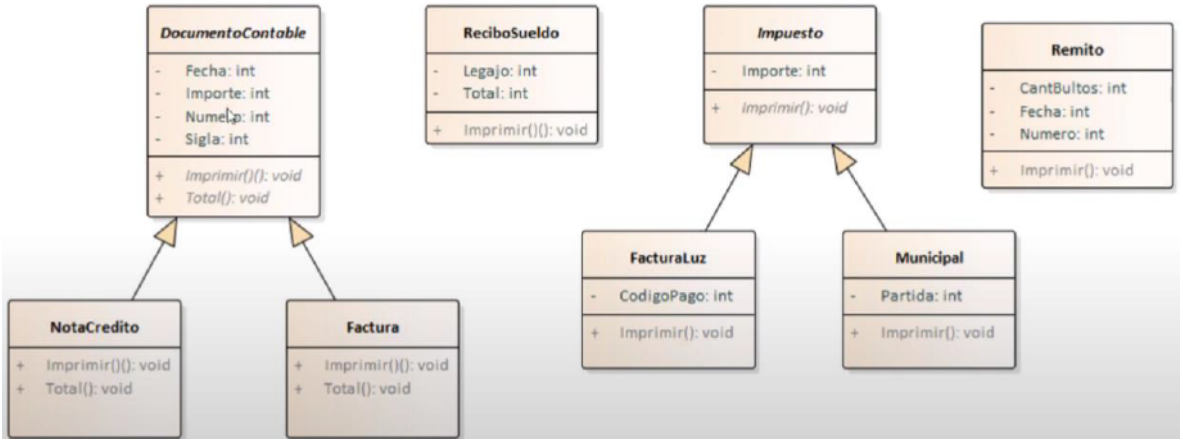

Figura 2.5: Clases de los diferentes documentos con la agregación del método imprimir.

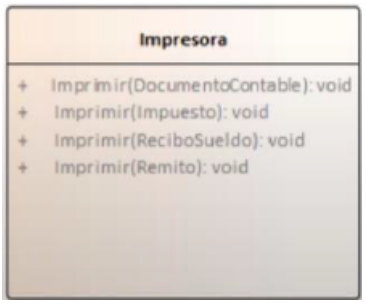

Figura 2.6: Clase que imprime los documentos con reducción de métodos.

Por último se observa que sigue siendo difícil continuar agregando documentos ya que todos tienen interfaces diferentes para imprimir aunque tengan el mismo nombre de operación, no son interfaces compatibles debido a que se encuentran en clases diferentes, además en la clase Impresora se debe seguir ajustando el método Imprimir. En base a esto se crea una clase interfaz denominada “Iimprimible”, que contiene el método “Imprimir”. Dicha interfaz será utilizada por todas las clases, entonces, en cada una se deberá desarrollar el método Imprimir adecuado (Figura 2.7). Específicamente en la clase Impresora, el método Imprimir recibe como parámetro un objeto de tipo “Iimprimible” ya que

si a una clase se le adjudica una interfaz se puede tratar como un objeto de tipo interfaz, en este caso de tipo IImprimible, dentro del método se llamará al método Imprimir del objeto recibido por parámetro, el cual imprimirá en pantalla lo que se desarrolle en su clase (Figura 2.8). Se obtiene como resultado final para poner en práctica, un diagrama en donde se visualiza la aplicación de la interfaz IImprimible en las clases Recibo de Sueldo, Remito, Documento Contable e Impuesto, estas dos últimas son a su vez clases abstractas y aplican herencia a Nota de Crédito y Factura y a Factura de Luz y Municipal respectivamente, también se observa que Impresora aplica la interfaz.

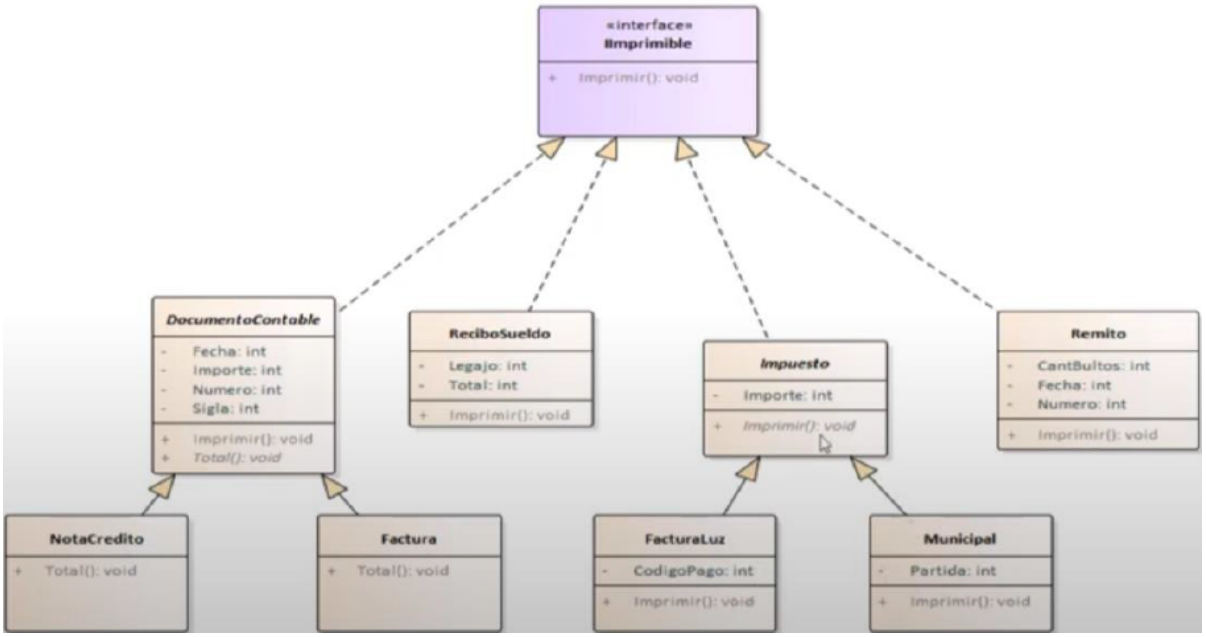

Figura 2.7: La interfaz IImprimible y las clases que la implementan.

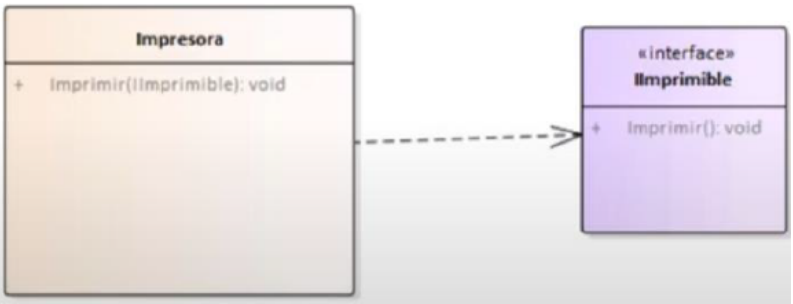

Figura 2.8: Clase Impresora implementando la clase IImprimible.

A continuación se expone el detalle en código de cada clase:

- Desarrollo de la clase padre abstracta Documento Contable:

```
5 referencias
internal  abstract class DocumentoContable : Iimprimible
{
    protected DateTime fecha;
    protected double importe;
    protected int numero;
    protected string sigla;

    2 referencias
    public DocumentoContable(string sigla, int numero, double importe)
    {
        fecha = DateTime.Now;
        this.sigla = sigla;
        this.numero = numero;
        this.importe = importe;
    }
```

Figura 3.1: Atributos y método constructor.

```
0 referencias
public DateTime getFecha() { return fecha; }
0 referencias
public void setFecha(DateTime fecha) { this.fecha = fecha; }

2 referencias
public double getImporte() { return importe; }
0 referencias
public void setImporte(double importe) { this.importe = importe; }

0 referencias
public int getNumero() { return numero; }
0 referencias
public void setNumero(int numero) { this.numero = numero; }

0 referencias
public string getSigla() { return sigla; }
0 referencias
public void setSigla(string sigla) { this.sigla = sigla; }

2 referencias
public  void Imprimir()
{
    Console.WriteLine($"Impresion de {sigla} de ${total()} del dia {fecha}");
}
3 referencias
public abstract double total();
```

Figura 3.2: Métodos *get* y *set* de los atributos, desarrollo del método Imprimir y el método abstracto Total.

- Desarrollo de la clase hija Factura:

```
internal class Factura : DocumentoContable
{
    1 referencia
    public Factura(string siglas, int numero, double importe) : base(siglas, numero, importe)
    {

    }
    2 referencias
    public override double total()
    {
        return (getImporte() * 1.21);

    }

}
```

Figura 3.3: Método constructor base y la funcionalidad del método heredado Total.

- Desarrollo de la clase hija Nota de Crédito:

```
internal class NotaCredito : DocumentoContable
{
    1 referencia
    public NotaCredito(string siglas, int numero, double importe) : base(siglas, numero, importe)
    {

    }
    2 referencias
    public override double total()
    {
        return (getImporte() * 1.21 * -1);

    }

}
```

Figura 3.4: Método constructor base y la funcionalidad del método heredado Total.

- Desarrollo de la clase Recibo de Sueldo:

```
internal class ReciboSueldo: Iimprimible
{
    protected int legajo;
    protected double total;
    1 referencia
    public ReciboSueldo(int legajo, double total) {
        this.legajo = legajo;
        this.total = total;
    }

    2 referencias
    public void Imprimir()
    {
        Console.WriteLine($"Impresion de Recibo de sueldo de: {legajo} con el total: ${total}");
    }
}
```

Figura 3.5: Sus atributos, método constructor y la funcionalidad del método Imprimir.

- Desarrollo de la clase padre abstracta Impuesto:

```
internal abstract class Impuesto: Iimprimible
{
    protected double importe;

    2 referencias
    public Impuesto(double importe)
    {
        this.importe = importe;
    }

    4 referencias
    public abstract void Imprimir();
}
```

Figura 3.6: Su atributo, método constructor y la sentencia del método Imprimir.

- Desarrollo de la clase hija Factura de Luz:

```
internal class FacturaLuz: Impuesto
{
    private int codigoPago;

    1 referencia
    public FacturaLuz(int codigoPago, double importe):base(importe)
    {
        this.codigoPago = codigoPago;
    }
    3 referencias
    public override void Imprimir()
    {
        Console.WriteLine($"Impresion de factura de luz {codigoPago} por el total de: ${importe}");
    }
}
```

Figura 3.7: Su atributo propio, el método constructor base y la definición del comportamiento del método Imprimir.

- Desarrollo de la clase hija Impuestos Municipales:

```
internal class Municipal:Impuesto
{
    private int partida;
    1 referencia
    public Municipal(int partida, double importe):base(importe)
    {
        this.partida = partida;
    }

    3 referencias
    public override void Imprimir()
    {
        Console.WriteLine($"Impresion de Impuestos Municipales n° {partida} por el total de ${importe}");
    }
}
```

Figura 3.8: Su atributo propio, el método constructor base y la implementación del método Imprimir.

- Desarrollo del método Remito:

```
internal class Remito:Iimprimible
{
    private int cantBultos;
    private DateTime fecha;
    private int numero;

    1 referencia
    public Remito(int cantBultos, int numero)
    {
        this.cantBultos = cantBultos;
        fecha = DateTime.Now;
        this.numero = numero;
    }
    2 referencias
    public void Imprimir()
    {
        Console.WriteLine($"Impresion de Remito: {numero} con {cantBultos} bultos del dia {fecha}");
    }
}
```

Figura 3.9: Atributos propios, el método constructo y la definición del método Imprimir.

- Desarrollo de la clase Impresora:

```
internal class Impresora
{
    6 referencias
    public void Imprimir(Iimprimible unImprimible)
    {
        unImprimible.Imprimir();
    }
}
```

Figura 3.10: Método Imprimir que recibe como parámetro un objeto de tipo Iimprimible y aplica en él el método Imprimir desarrollado en su clase.

- Desarrollo de la clase interfaz Iimprimible:

```
internal interface Iimprimible
{
    7 referencias
    void Imprimir();
}
```

Figura 3.11: Método Imprimir.

3. Resultados

Se logró un sistema capaz de realizar las tareas requeridas efectivamente que cuenta con cierta facilidad de mantenimiento e implementación de nuevos componentes en el futuro.

A continuación se muestra parte del sistema en consola de Windows:

- Menú principal:

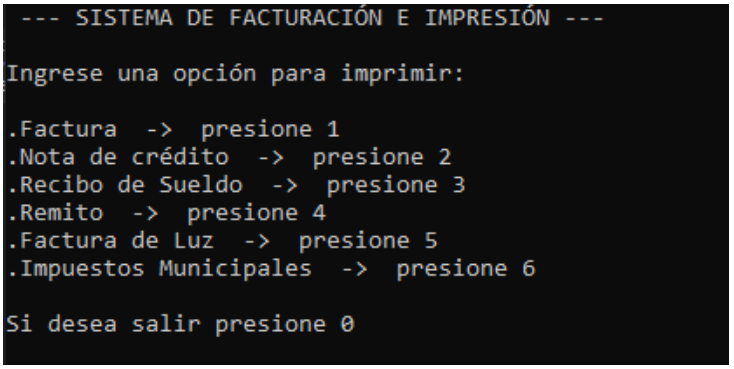

Figura 3.12: Menú principal.

- Selección de ejemplo Nota de crédito:

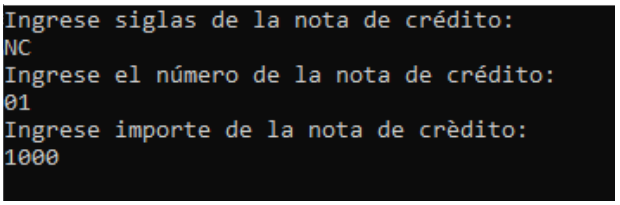

Figura 3.13: Pedido de datos para la impresión.

- Impresión final:

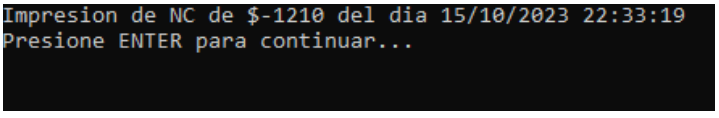

Figura 3.14: Impresión final (Método Total aplicado).

4. Conclusiones

En cuestión de solucionar el problema planteado consideramos que se utilizaron las herramientas como herencia, clases abstractas e interfaces de forma correcta y sencilla. Si bien no hubo mayor complejidad, nos costó entender e integrar concretamente la interfaz, a pesar de eso pudimos realizarlo con éxito y a su vez llevándonos la experiencia para futuros trabajos.

Este ejercicio logró ilustrar claramente a través del proceso de cambios que se fueron realizando, la utilidad y ventajas del uso de interfaces.

Un punto para mejorar podría ser la implementación de un método Fotocopiar en la interfaz IImprimible, en donde en la clase Impresora este método recibe como parámetro un objeto de tipo IImprimible, en teoría ya creado, y a través del método desarrollado en cada clase se cree otro guardándolo en un lista de objetos. Concluimos que como es nuestra primera vez trabajando con interfaces se logró llevar a término el proyecto de manera efectiva, entendiendo y observando en la práctica el funcionamiento del concepto interfaces.

5. Referencias

[1] - https://desarrolloweb.com/articulos/499.php

[2] - https://learn.microsoft.com/es-es/dotnet/csharp/fundamentals/object-oriented/inheritance

[3] - https://estradawebgroup.com/Post/Que-son-y-como-implementar-interfaces-en-C/20668

[4] - https://www.glosarioit.com/Clase_abstracta

[5] - https://www.edrawsoft.com/es/example-uml-class-diagram.html

[6] - https://www.edrawsoft.com/es/article/class-diagram-relationships.html